

# 24th International Conference on Principles of Distributed Systems

OPODIS 2020, December 14–16, 2020, Strasbourg, France  
(Virtual Conference)

Edited by

Quentin Bramas  
Rotem Oshman  
Paolo Romano



*Editors*

**Quentin Bramas** 

University of Strasbourg, ICUBE, CNRS, Strasbourg, France  
bramas@unistra.fr

**Rotem Oshman**

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

**Paolo Romano** 

Lisbon University & INESC-ID, Portugal  
romano@inesc-id.pt

*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 978-3-95977-176-4**

*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/978-3-95977-176-4>.

*Publication date*

January, 2021

*Bibliographic information published by the Deutsche Nationalbibliothek*

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

*License*

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



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

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

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPICs.OPODIS.2020.0

ISBN 978-3-95977-176-4

ISSN 1868-8969

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

## LIPICs – Leibniz International Proceedings in Informatics

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

### *Editorial Board*

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

**ISSN 1868-8969**

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





## ■ Contents

Preface	
<i>Quentin Bramas, Rotem Oshman, and Paolo Romano</i> .....	0:ix
Program Committee	
.....	0:xi–0:xii
Steering Committee	
.....	0:xiii
External Reviewers	
.....	0:xv–0:xvi

### Invited Talks

Big Data Processing: Security and Scalability Challenges	
<i>Pascal Felber</i> .....	1:1–1:1
Byzantine Agreement and SMR with Sub-Quadratic Message Complexity	
<i>Idit Keidar</i> .....	2:1–2:1
Can We Automate Our Own Work – or Show That It Is Hard?	
<i>Jukka Suomela</i> .....	3:1–3:1

### Regular Papers

Byzantine Lattice Agreement in Asynchronous Systems	
<i>Xiong Zheng and Vijay Garg</i> .....	4:1–4:16
Heterogeneous Paxos	
<i>Isaac Sheff, Xinwen Wang, Robbert van Renesse, and Andrew C. Myers</i> .....	5:1–5:17
Multi-Threshold Asynchronous Reliable Broadcast and Consensus	
<i>Martin Hirt, Ard Kastrati, and Chen-Da Liu-Zhang</i> .....	6:1–6:16
Echo-CGC: A Communication-Efficient Byzantine-Tolerant Distributed Machine Learning Algorithm in Single-Hop Radio Network	
<i>Qinzi Zhang and Lewis Tseng</i> .....	7:1–7:16
AKSEL: Fast Byzantine SGD	
<i>Amine Boussetta, El-Mahdi El-Mhamdi, Rachid Guerraoui, Alexandre Maurer, and Sébastien Rouault</i> .....	8:1–8:16
ACE: Abstract Consensus Encapsulation for Liveness Boosting of State Machine Replication	
<i>Alexander Spiegelman, Arik Rinberg, and Dahlia Malkhi</i> .....	9:1–9:18
Security Analysis of Ripple Consensus	
<i>Ignacio Amores-Sesar, Christian Cachin, and Jovana Mićić</i> .....	10:1–10:16
Information Theoretic HotStuff	
<i>Ittai Abraham and Gilad Stern</i> .....	11:1–11:16

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano



Leibniz International Proceedings in Informatics

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

Rational Behaviors in Committee-Based Blockchains <i>Yackolley Amoussou-Guenou, Bruno Biais, Maria Potop-Butucaru, and Sara Tucci-Piergiovanni</i> .....	12:1–12:16
Relaxed Queues and Stacks from Read/Write Operations <i>Armando Castañeda, Sergio Rajsbaum, and Michel Raynal</i> .....	13:1–13:19
Fast and Space-Efficient Queues via Relaxation <i>Dempsey Wade and Edward Talmage</i> .....	14:1–14:16
Recoverable, Abortable, and Adaptive Mutual Exclusion with Sublogarithmic RMR Complexity <i>Daniel Katzan and Adam Morrison</i> .....	15:1–15:16
Optimal Resilience in Systems That Mix Shared Memory and Message Passing <i>Hagit Attiya, Sweta Kumari, and Noa Schiller</i> .....	16:1–16:16
CSR++: A Fast, Scalable, Update-Friendly Graph Data Structure <i>Soukaina Firmlı, Vasileios Trigonakis, Jean-Pierre Lozi, Iraklis Psaroudakis, Alexander Weld, Dalila Chiadmi, Sungpack Hong, and Hassan Chafi</i> .....	17:1–17:16
Locally Solvable Tasks and the Limitations of Valency Arguments <i>Hagit Attiya, Armando Castañeda, and Sergio Rajsbaum</i> .....	18:1–18:16
Approximate Majority with Catalytic Inputs <i>Talley Amir, James Aspnes, and John Lazarsfeld</i> .....	19:1–19:16
Distributed Runtime Verification Under Partial Synchrony <i>Ritam Ganguly, Anik Momtaz, and Borzoo Bonakdarpour</i> .....	20:1–20:17
Decentralized Runtime Enforcement of Message Sequences in Message-Based Systems <i>Mahboubeh Samadi, Fatemeh Ghassemi, and Ramtin Khosravi</i> .....	21:1–21:18
Broadcasting Competitively Against Adaptive Adversary in Multi-Channel Radio Networks <i>Haimin Chen and Chaodong Zheng</i> .....	22:1–22:16
Dynamic Byzantine Reliable Broadcast <i>Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, and Andrei Tonkikh</i> .....	23:1–23:18
Broadcasting with Mobile Agents in Dynamic Networks <i>Shantanu Das, Nikos Giachoudis, Flaminia L. Luccio, and Euripides Markou</i> ....	24:1–24:16
On Broadcast in Generalized Network and Adversarial Models <i>Chen-Da Liu-Zhang, Varun Maram, and Ueli Maurer</i> .....	25:1–25:16
Maximally Resilient Replacement Paths for a Family of Product Graphs <i>Mahmoud Parham, Klaus-Tycho Foerster, Petar Kosic, and Stefan Schmid</i> .....	26:1–26:16
Self-Stabilizing Byzantine-Resilient Communication in Dynamic Networks <i>Alexandre Maurer</i> .....	27:1–27:11
Fast Deterministic Algorithms for Highly-Dynamic Networks <i>Keren Censor-Hillel, Neta Dafni, Victor I. Kolobov, Ami Paz, and Gregory Schwartzman</i> .....	28:1–28:16

Approximating Bipartite Minimum Vertex Cover in the CONGEST Model <i>Salwa Faour and Fabian Kuhn</i> .....	29:1–29:16
Distributed Distance Approximation <i>Bertie Ancona, Keren Censor-Hillel, Mina Dalirrooyfard, Yuval Efron, and Virginia Vassilevska Williams</i> .....	30:1–30:17
Fast Hybrid Network Algorithms for Shortest Paths in Sparse Graphs <i>Michael Feldmann, Kristian Hinnenthal, and Christian Scheideler</i> .....	31:1–31:16
Secured Distributed Algorithms Without Hardness Assumptions <i>Leonid Barenboim and Harel Levin</i> .....	32:1–32:16
Uniform Bipartition in the Population Protocol Model with Arbitrary Communication Graphs <i>Hiroto Yasumi, Fukuhito Ooshita, Michiko Inoue, and Sébastien Tixewil</i> .....	33:1–33:16



## ■ Preface

The papers in this volume were presented at the 24th International Conference on Principles of Distributed Systems (OPODIS 2020), held on December 14–16, 2020. Originally planned to be held in Strasbourg, France, the conference was held online due to the COVID19 pandemic.

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 75 submissions, each of which underwent a double-blind peer review process, by at least three members of the Program Committee with the help of external reviewers. Overall, the quality of the submissions was very high. From the 75 submissions, 30 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.

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano



Leibniz International Proceedings in Informatics

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

The Best Paper Award was awarded to Salwa Faour and Fabian Kuhn for their paper titled “Approximating Bipartite Minimum Vertex Cover in the CONGEST model”. The Best Student Paper Award was given to Amine Boussetta for his paper titled “Fast Byzantine SGD”, co-authored with Rachid Guerraoui, Alexandre Maurer and Sébastien Rouault.

This year OPODIS had three distinguished invited keynote speakers: Idit Keidar (Technion), Jukka Suomela (Aalto University) and Pascal Felber (University of Neuchâtel).

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 2020

Quentin Bramas (University of Strasbourg, ICUBE, France)

Rotem Oshman (Technion, Israel)

Paolo Romano (University of Lisbon and INESC-ID, Portugal)

## ■ Program Committee

### General Chair

Quentin Bramas, University of Strasbourg, France

### Program Chairs

Rotem Oshman, Technion, Israel

Paolo Romano, University of Lisbon and INESC-ID, Portugal

### Program Committee

Silvia Bonomi, University of Rome Sapienza, Italy

Dave Dice, Oracle, USA

Diego Didona, IBM Research Zurich, Switzerland

Aleksandar Drakojevic, Microsoft Research, UK

Laurent Feuilloley, Universidad de Chile, Chile

Vijay Garg, UT Austin, USA

Wojciech Golab, University of Waterloo, Canada

Taisuke Izumi, Nagoya Institute of Technology, Japan

Shir Landau, Princeton University, USA

João Leitão, Universidade Nova de Lisboa, Portugal

Othon Michail, University of Liverpool, UK

Pedro Montealegre, Universidad Adolfo Ibáñez, Chile

Fukuhito Ooshita, Nara Institute of Science and Technology, Japan

Roberto Palmieri, LeHigh University, USA

Marta Patino, Universidad Politécnica de Madrid, Spain

Fernando Pedone, University of Lugano, Switzerland

Sebastiano Peluso, Facebook, USA

Francesco Quaglia, University of Rome Tor Vergata, Italy

Vivien Quema, Grenoble INP, France

Luís Rodrigues, University of Lisbon & INESC-ID, Portugal

William Rosenbaum, MPI Saarbrücken, Germany

Nicola Santoro, Carleton University, Canada

Jared Saia, University of New Mexico, USA

Valerio Schiavoni, University of Neuchâtel, Switzerland

Stefan Schmid, University of Vienna, Austria

Ulrich Schmid, Vienna University of Technology, Austria

Pierre Sutra, Télécom SudParis, France

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano



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

**0:xii    Program Committee**

Hsin-Hao Su, Boston College, USA

Jara Uitto, Aalto University, Finland

Philipp Woelfel, University of Calgary, Canada



## ■ Steering Committee

Pascal Felber, University of Neuchâtel, Switzerland (Chair)

Faith Ellen, University of Toronto, Canada

Luis Rodrigues, University of Lisbon & IST, Portugal

Roy Friedman, Technion, Israel

Seth Gilbert, National University of Singapore

Yukiko Yamauchi, Kyushu University, Japan

Xavier Defago, Tokyo Institute of Technology, Japan

Youla Fatourou, University of Crete, Greece



## ■ External Reviewers

Sander Aarts, Aalto University  
Ittai Abraham, VMWare Research  
Yehuda Afek, Tel-Aviv University  
Abdullah Almethen, University of Liverpool  
Saeed Amiri, University of Cologne  
Philipp Bamberger, University of Freiburg  
Ezio Bartocci, TU Wien  
Benyamin Bashari, University of Calgary  
Sebastien Bouchard, Université du Québec en Outaouais  
David Chan, University of Calgary  
Yi-Jun Chang, ETH Zurich  
Davide Cingolani, Lockless Srl  
Andrea Clementi, Università degli Studi di Roma "TorVergata"  
Matthew Connor, University of Liverpool  
Francesco d'Amore, Université Côte D'Azur  
Pedro Fouto, NOVALINCS & DI-FCT-UNL  
Yanni Georghiades, University of Texas at Austin  
Ahmed Hassan, Lehigh University  
Changyong Hu, University of Texas at Austin  
Ernesto Jiménez, Universidad Politécnica de Madrid  
Akinori Kawachi, Mie University  
Yonghwan Kim, Nagoya Institute of Technology  
Naoki Kitamura, Nagoya Institute of Technology  
Petr Kuznetsov, Institut Polytechnique de Paris  
Mikel Larrea, University of the Basque Country  
Rustam Latypov, Aalto University  
Chunyu Mao, University of Waterloo  
Romolo Marotta, Lockless Srl  
Hammurabi Mendes, Davidson College  
Claudio Mezzina, Università degli Studi di Urbino  
Jaroslaw Mirek, University of Wroclaw  
Doron Mukhtar, Tel Aviv University  
Junya Nakamura, Toyohashi University of Technology  
Dominik Pajak, Wroclaw University of Science and Technology  
Alessandro Pellegrini, University of Rome Tor Vergata

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano



Leibniz International Proceedings in Informatics

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

**0:xvi External Reviewers**

Julian Portmann, ETH Zurich  
Ivan Rapaport, Universidad de Chile  
Hugo Rincon Galeana, TU Wien  
Isabelly Rocha, University of Neuchatel  
Joel Rybicki, IST Austria  
Thomas Schlögl, TU Wien  
Carlos Segarra, Imperial College London  
Eric Severson, UC Davis  
Masahiro Shibata, Kyushu Institute of Technology  
Sebastian Siebertz, University of Bremen  
George Skretas, University of Liverpool  
Robert Streit, University of Texas at Austin  
Jan Studeny, Aalto University  
Hao Tan, University of Waterloo  
Michail Theofilatos, University of Liverpool  
Lewis Tseng, Boston College  
Przemek Uznanski, University of Wrocław  
Nitin Vaidya, Georgetown University  
Marko Vukolić, IBM Research, Switzerland  
Kyrill Winkler, TU Wien  
Maxwell Young, Mississippi State University  
Xiong Zheng, University of Texas at Austin

# Big Data Processing: Security and Scalability Challenges

Pascal Felber

University of Neuchâtel, Switzerland

---

## Abstract

The processing of large amounts of data requires significant computing power and scalable architectures. This trend makes the use of Cloud computing and off-premises data centres particularly attractive, but exposes companies to the risk of data theft. This is a key challenge toward exploiting public Clouds, as data represents for many companies their most valuable asset. In this talk, we will discuss about mechanisms to ensure secure and privacy-preserving Big Data processing on computing architectures supporting horizontal and vertical scalability.

**2012 ACM Subject Classification** Computer systems organization → Cloud computing; Security and privacy → Privacy-preserving protocols

**Keywords and phrases** Big Data

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2020.1

**Category** Invited Talk



© Pascal Felber;

licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramer, Rotem Oshman, and Paolo Romano; Article No. 1; pp. 1:1–1:1

Leibniz International Proceedings in Informatics



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



# Byzantine Agreement and SMR with Sub-Quadratic Message Complexity

Idit Keidar

Technion, Haifa, Israel

---

## Abstract

Byzantine Agreement (BA) has been studied for four decades by now, but until recently, has been considered at a fairly small scale. In recent years, however, we begin to see practical use-cases of BA in large-scale systems, which motivates a push for reduced communication complexity. Dolev and Reischuk's well-known lower bound stipulates that any deterministic algorithm requires  $\Omega(n^2)$  communication in the worst-case, and until fairly recently, almost all randomized algorithms have had at least quadratic complexity as well. This talk will present two new algorithms breaking this barrier.

The first part of the talk will consider a fully asynchronous setting, focusing on randomized BA whose safety and liveness guarantees hold with high probability. It will present the first asynchronous Byzantine Agreement algorithm with sub-quadratic communication complexity. This algorithm exploits VRF-based committee sampling, which it adapts for the asynchronous model.

The second part of the talk will consider the eventually synchronous model, where BA and State Machine Replication (SMR) can be solved with deterministic safety and liveness guarantees. In this context, randomization is used in order to reduce the expected communication complexity. The talk will present an algorithm for round synchronization, which is a building block for BA and SMR and constitutes the main performance bottleneck therein. It will present an algorithm that, for the first time, achieves round synchronization with expected linear message complexity and expected constant latency. Existing protocols can use this round synchronization algorithm to solve Byzantine SMR with the same asymptotic performance.

The first part of the talk is based on joint work with Shir Cohen and Alexander Spiegelman, and the second part of the talk is based on joint work with Oded Naor.

**2012 ACM Subject Classification** Networks → Network algorithms; Computing methodologies → Distributed algorithms

**Keywords and phrases** Distributed Computing, Byzantine Agreement

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2020.2

**Category** Invited Talk

**Related Version** This talk covers results from [1], <https://doi.org/10.4230/LIPIcs.DISC.2020.25> and [2], <https://doi.org/10.4230/LIPIcs.DISC.2020.26>.

---

## References

- 1 Shir Cohen, Idit Keidar, and Alexander Spiegelman. Not a coincidence: Sub-quadratic asynchronous byzantine agreement WHP. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, volume 179 of *LIPIcs*, pages 25:1–25:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.DISC.2020.25.
- 2 Oded Naor and Idit Keidar. Expected linear round synchronization: The missing link for linear byzantine SMR. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, volume 179 of *LIPIcs*, pages 26:1–26:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.DISC.2020.26.



© Idit Keidar;

licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 2; pp. 2:1–2:1

Leibniz International Proceedings in Informatics



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





# Can We Automate Our Own Work – or Show That It Is Hard?

Jukka Suomela 

Aalto University, Finland

<https://jukkasuomela.fi/>

[jukka.suomela@aalto.fi](mailto:jukka.suomela@aalto.fi)

---

## Abstract

Computer scientists seek to understand what can be automated, but what do we know about automating our own work? Can we outsource our own research questions to computers? In this talk I will discuss this question from the perspective of the theory of distributed computing. I will present not only recent examples of human-computer-collaborations that have resulted in major breakthroughs in our understanding of distributed computing, but I will also explore the fundamental limits of such approaches.

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

**Keywords and phrases** Distributed Computing

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2020.3

**Category** Invited Talk



© Jukka Suomela;

licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 3; pp. 3:1–3:1

Leibniz International Proceedings in Informatics



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



# Byzantine Lattice Agreement in Asynchronous Systems

Xiong Zheng

Electrical and Computer Engineering, University of Texas at Austin, TX, USA

Vijay Garg

Electrical and Computer Engineering, University of Texas at Austin, TX, USA

---

## Abstract

We study the Byzantine lattice agreement (BLA) problem in asynchronous distributed message passing systems. In the BLA problem, each process proposes a value from a join semi-lattice and needs to output a value also in the lattice such that all output values of correct processes lie on a chain despite the presence of Byzantine processes. We present an algorithm for this problem with round complexity of  $O(\log f)$  which tolerates  $f < \frac{n}{5}$  Byzantine failures in the asynchronous setting without digital signatures, where  $n$  is the number of processes. This is the first algorithm which has logarithmic round complexity for this problem in asynchronous setting. Before our work, Di Luna et al give an algorithm for this problem which takes  $O(f)$  rounds and tolerates  $f < \frac{n}{3}$  Byzantine failures. We also show how this algorithm can be modified to work in the authenticated setting (i.e., with digital signatures) to tolerate  $f < \frac{n}{3}$  Byzantine failures.

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

**Keywords and phrases** Byzantine Lattice Agreement, Asynchronous

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2020.4

**Related Version** Full version hosted on <https://arxiv.org/abs/2002.06779>.

**Funding** *Vijay Garg*: This work was supported in parts by the National Science Foundation Grants CNS-1812349, CNS-1563544, and the Cullen Trust Endowed Professorship.

## 1 Introduction

In distributed systems, reaching agreement in the presence of process failures is a fundamental task. Understanding the kind of agreement that can be reached helps us understand the limitation of distributed systems with failures. Consensus [15] is the most fundamental problem in distributed computing. In this problem, each process proposes some input value and has to decide on some output value such that all correct processes decide on the same valid output. In synchronous message systems with crash failures, consensus cannot be solved in fewer than  $f + 1$  rounds [9]. In asynchronous systems, consensus is impossible in the presence of even one crash failure [11]. The  $k$ -set agreement [5] is a generalization of consensus, in which processes can decide on at most  $k$  values instead of just one single value. The  $k$ -set agreement cannot be solved in asynchronous systems if the number of crash failures  $f \geq k$  [3, 12]. The paper [6] shows that  $k$ -set agreement problem cannot be solved by less than  $\lfloor \frac{f}{k} \rfloor$  rounds if  $n \geq f + k + 1$  in crash failure model. The lattice agreement problem was proposed by Attiya et al [1] to solve the atomic snapshot object problem in shared memory systems. In this problem, each process  $i \in [n]$  has input  $x_i$  and needs to output  $y_i$  such that the following properties are satisfied. 1) **Downward-Validity**:  $x_i \leq y_i$  for each correct process  $i$ . 2) **Upward-Validity**:  $y_i \leq \sqcup \{x_i \mid i \in [n]\}$ . 3) **Comparability**: for any two correct processes  $i$  and  $j$ , either  $y_i \leq y_j$  or  $y_j \leq y_i$ .



© Xiong Zheng and Vijay Garg;

licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 4; pp. 4:1–4:16

Leibniz International Proceedings in Informatics



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

Attiya et al in [1] present a generic algorithm to transform any protocol for the lattice agreement problem to a protocol for implementing an atomic snapshot object in shared memory systems. This transformation can be easily implemented in message passing systems by replacing each read and write step with sending “*read*” and “*write*” messages to all and waiting for acknowledgements from  $n - f$  different processes. Conversely, if we can implement an atomic snapshot object, lattice agreement can also be solved easily both on shared-memory and message passing systems with only crash failures. Thus, solving the lattice agreement problem in message passing systems is equivalent to implementing an atomic snapshot object in message passing systems with only crash failures.

Using lattice agreement protocols, Faleiro et al [10] give procedures to build a special class of linearizable and serializable replicated state machines which only support query operations and update operations but not mixed query-update operations. Later, Xiong et al [19] propose some optimizations for their procedure for implementing replicated state machines from lattice agreement in practice. They propose a method to truncate the logs maintained in the procedure in [10]. The recent paper [17] by Skrzypczak et al proposes a protocol based on generalized lattice agreement [10], which is a multi-shot version of lattice agreement problem, to provide linearizability for state based conflict-free data types [16].

In message passing systems with crash failures, the lattice agreement problem is well studied [1, 19, 20, 13]. The best upper bound for both synchronous systems and asynchronous systems is  $O(\log f)$  rounds. In the Byzantine failure model, a variant of the lattice agreement problem is first studied by Nowak et al [14]. Then, Di Luna et al [8] propose a validity condition which still permits the application of lattice agreement protocol in obtaining atomic snapshots and implementing a special class of replicated state machines. They present an  $O(f)$  rounds algorithm for the Byzantine lattice agreement problem in asynchronous message systems. For synchronous message systems, a recent preprint by Xiong et al [18] gives three algorithms. The first algorithm takes  $O(\sqrt{f})$  rounds and has the early stopping property. The second and third algorithm takes  $O(\log n)$  and  $O(\log f)$  rounds but are not early stopping. All three algorithms can tolerate  $f < \frac{n}{3}$  failures. They also show how to modify their algorithms to work for authenticated settings and tolerates  $f < \frac{n}{2}$  failures. The preprint by Di Luna et al [7] presents an algorithm which takes  $O(\log f)$  rounds and tolerates  $f < \frac{n}{4}$  failures and shows how to improve resilience to  $f < \frac{n}{3}$  by using digital signatures.

In this work, we present new algorithms for the Byzantine lattice agreement (BLA) problem in asynchronous message systems. In this problem, each process  $i \in [n]$  has input  $x_i$  from a join semi-lattice  $(X, \leq, \sqcup)$  with  $X$  being the set of elements in the lattice,  $\leq$  being the partial order defined on  $X$ , and  $\sqcup$  being the join operation. The lattice can be infinite. Each process  $i$  has to output some  $y_i \in X$  such that the following properties are satisfied. Let  $C$  denote the set of correct processes in the system and  $f_a$  denote the actual number of Byzantine processes in the system.

**Comparability:** For all  $i \in C$  and  $j \in C$ , either  $y_i \leq y_j$  or  $y_j \leq y_i$ .

**Downward-Validity:** For all  $i \in C$ ,  $x_i \leq y_i$ .

**Upward-Validity:**  $\sqcup\{y_i \mid i \in C\} \leq \sqcup(\{x_i \mid i \in C\} \cup B)$ , where  $B \subset X$  and  $|B| \leq f_a$ .

The first two requirements are straightforward. Upward-Validity requires that the total number of values that can be introduced by Byzantine processes into the decision value of correct processes can be at most the number of actual Byzantine processes in the system. One may argue that if a Byzantine process proposes the largest element of the input lattice, then correct processes may always decide on the largest element. For applications, we can impose an additional constraint on the initial proposal of all processes. In the case of a Boolean lattice, we can require that the initial proposal for any process must be a singleton. More generally, we can impose the requirement that the initial proposal of any process must have the height less than some constant.

Our contribution is summarized in Table 1. First, we present an algorithm for the BLA problem in asynchronous systems without the digital signatures assumption which takes  $O(\log f)$  rounds and  $f < \frac{n}{5}$ . The algorithm achieves exponential improvement in round complexity compared to the previous best algorithm in [8]. Then, we show how to improve the resilience to  $f < \frac{n}{3}$  with the digital signatures assumption. The round complexity of our algorithm matches the best round complexity achieved in synchronous model [18].

■ **Table 1** Our Results.

Model	Digital Signatures?	Reference	Rounds	Resilience
Synchronous	No	[18]	$O(\log f)$	$f < \frac{n}{3}$
	Yes		$O(\log f)$	$f < \frac{n}{2}$
Asynchronous	No	[8]	$O(f)$	$f < \frac{n}{3}$
	No	This paper	$O(\log f)$	$f < \frac{n}{5}$
	Yes			$f < \frac{n}{3}$

## 2 System Model

We assume a distributed asynchronous message system with  $n$  processes with unique ids in  $\{1, 2, \dots, n\}$ . The communication graph is completely connected, i.e., each process can send messages to any other process in the system. We assume that the communication channel between any two processes is reliable. There is no upper bound on message delay. We assume that processes can have Byzantine failures but at most  $f < n/3$  processes can be Byzantine in any execution of the algorithm. We use parameter  $f_a$  to denote the actual number of Byzantine processes in a system. By our assumption, we must have  $f_a \leq f$ . Byzantine processes can deviate arbitrarily from the algorithm. We say a process is correct or non-faulty if it is not a Byzantine process. We consider both systems with and without digital signatures. In a system with digital signatures, Byzantine processes cannot forge the signature of correct processes.

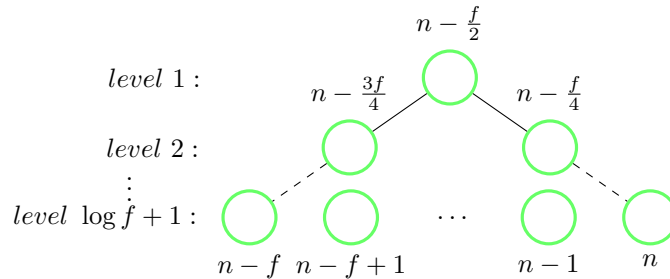
## 3 Algorithm for the Asynchronous model without Digital Signatures

In this section, we present an algorithm for the BLA problem in asynchronous systems which takes  $O(\log f)$  rounds of asynchronous communication and tolerates  $f < \frac{n}{5}$  Byzantine failures. Our algorithm applies a recursive approach similar to the algorithms designed for crash failure model in [20], which is inspired by the algorithm in [2] designed for atomic snapshot objects in shared memory systems. The high level idea of the recursive approach is to apply a classifier procedure to divide a group of processes into the slave subgroup and the master subgroup and update their values such that the values of the slave group is less than the values of the master group. Then, by recursively applying such a classifier procedure within each subgroup, eventually all processes have comparable values. In crash failure model, the classifier procedure only needs to guarantee the following two properties: (C1) The value of a correct slave process is at most the value of any correct master process, (C2) The size of the union of all values of correct slave processes is at most  $k$ , which is a threshold parameter associated with the classifier procedure and serves as knowledge threshold.

Suppose we have a classifier procedure in the crash failure model with properties (C1) and (C2). The binary tree in Fig. 1 shows how processes invoke the classifier procedure recursively. Each node in the tree represents a classifier procedure with its threshold parameter  $k$  shown

#### 4:4 Byzantine Lattice Agreement in Asynchronous Systems

above the node. Before all processes traverse the tree and recursively invoke the classifier procedures along the way, an initial round is used to let all processes exchange their input values. After the initial round, each process obtains at least  $n - f$  values. The threshold parameters of classifier procedures in the tree are set in a binary way with low equal to  $n - f$  and high equal to  $n$ . The threshold parameter of the classifier procedure at the root node is set as  $n - \frac{f}{2}$ . Given a node with threshold parameter equal to  $k$ , the threshold parameter of its left child node and right child node are set as  $k - \frac{f}{2^{r+1}}$  and  $k + \frac{f}{2^{r+1}}$ , respectively. Then, all processes traverse the binary tree starting from the root and invoke the classifier procedures along the way. After a specific classifier procedure invocation, processes classified as slave traverses to the left subtree and processes classified as master traverse to the right subtree. We can observe that all labels in the binary tree up to level  $\log f$  are unique. The above properties (C1) and (C2) of the classifier procedure and our method to set the threshold parameter of each classifier procedure in the tree guarantee that 1) at level  $\log f + 1$ , processes in different nodes have comparable values, 2) at level  $\log f + 1$ , processes within the same node must have the same value. This will be formally proved when we present our algorithm.



■ **Figure 1** The Classification Tree.

In presence of Byzantine processes, (C1) and (C2) are not enough for recursively applying such classifier procedure within each subgroup. A Byzantine process in a slave group can introduce new values which are not known by some master process. To prevent that from happening, we introduce the notion of admissible values for a group (to be formally defined later), which is the set of values that processes in this group can ever have. We present a Byzantine tolerant classifier procedure with threshold parameter  $k$  which provides the following properties: (B1) Each correct slave process has  $\leq k$  values and each correct master process has  $> k$  values. (B2) The admissible values of the slave group is a subset of the value of any correct master process. (B3) The union of all admissible values in the slave group has size  $<$  the threshold parameter  $k$ .

Suppose now we have a Byzantine tolerant classifier which guarantees the above properties. The main algorithm, shown in Fig. 2, proceeds in asynchronous rounds. The Byzantine tolerant classifier procedure can take multiple rounds. For ease of presentation, we call each round in the classifier as a subround. Each process  $i$  maintains a value set  $V_i$  which contains a set of values and is updated at each round by invoking the classifier procedure. Each process  $i$  has a label  $l_i$ , which is used as the threshold parameter when it invokes the classifier procedure. Initially, each process has the same label  $k_0 = n - \frac{f}{2}$ . The label of a process is updated at each round according to the classification tree. Each process  $i$  also keeps track of a map  $S_i$ , which we call *the safe value map*.  $S_i[k]$  denotes the set of values that process  $i$  considers valid for label  $k$ . This safe value map is used by process  $i$  to restrict the admissible values of a group. In the main algorithm, a process uses the reliable broadcast primitive defined by Bracha [4] to send its value. When process  $i$  receives a value

broadcast by process  $j$  that is not in  $S_i[j]$ , it will not send echo this value to other processes. In the reliable broadcast primitive, a process uses **RB\_broadcast** to send a message and uses **RB\_broadcast** to reliably deliver a message. This primitive guarantees many nice properties. In our algorithm, we need the following two main properties: 1) If a message is reliably delivered by some correct process, then this message will eventually be reliably delivered by each correct process. 2) If a correct process reliably delivers a message from process  $p$ , then each correct process reliably delivers the same message from  $p$ .

<p><b>Code for process <math>i</math>:</b></p> <p><math>x_i</math>: input value    <math>y_i</math>: output value  <math>l_i</math>: label of process <math>i</math>. Initially, <math>l_i = k_0 = n - \frac{f}{2}</math>:  <math>V_i^r</math>: value set held by process <math>i</math> at round <math>r</math> of the algorithm  Map <math>S_i</math>: <math>S_i[k]</math> denote the safe value set for group <math>k</math></p> <p>/* Initial Round */  <b>1: RB_broadcast</b>(<math>x_i</math>), wait for <math>n - f</math> <b>RB_deliver</b>(<math>x_j</math>) from <math>p_j</math>  <b>2:</b> Set <math>V_i^1</math> as the set of values reliably delivered  /* Round 1 to <math>\log f</math> */  <b>3: for</b> <math>r := 1</math> to <math>\log f</math>  <b>4:</b>    (<math>V_i^{r+1}, class</math>) := <i>Classifier</i>(<math>V_i^r, l_i, r</math>)  <b>5:</b>    <b>if</b> <math>class = master</math>    <b>then</b> <math>l_i := l_i + \frac{f}{2^{r+1}}</math>  <b>6:</b>    <b>else</b>    <math>l_i := l_i - \frac{f}{2^{r+1}}</math>  <b>7: end for</b>  <b>8:</b> <math>y_i := \sqcup\{v \in V_i^{\log f + 1}\}</math></p> <p><b>Upon RB_deliver</b>(<math>x_j</math>) from <math>p_j</math>  <math>S_i[k_0] := S_i[k_0] \cup x_j</math></p>
---

■ **Figure 2**  $O(\log f)$  Rounds Algorithm for the BLA Problem.

In the initial round at lines 1-2, process  $i$  **RB\_broadcast** its input  $x_i$  to all and waits for **RB\_deliver** from  $n - f$  different processes. Then, it updates its value set to be the set of values reliably delivered at this round. When reliable delivering a value, process  $i$  adds this value into its safe value set for the initial group  $k_0 = n - \frac{f}{2}$ . The reliable delivery procedure is assumed to be running in background. So, the safe value set for the initial group keeps growing. By the properties of reliable broadcast, this safe value set can only contain at most one value from each process. This is used to ensure **Upward-Validity**.

After the initial round, we can assume that all values in the initial safe value set of each process are unique, which can be done by associating the sender's id with the value. At line 3-8, process  $i$  executes the classifier procedure (to be presented later) for  $\log f$  rounds. At each round, it invokes the classifier procedure to decide whether it is classified as a slave or a master and then updates its value accordingly. At round  $r$ , if process  $i$  is a master, it updates its label to be  $l_i := l_i + \frac{f}{2^{r+1}}$ . Otherwise, it updates its label to be  $l_i := l_i - \frac{f}{2^{r+1}}$ .

By applying properties (B1)-(B3), we can show that any two correct process  $i$  and  $j$  in the same group at the end of round  $\log f$  must have the same set of values. For any two processes in different group, by recursively applying property (B2), the values of one process must be subset of the values of the other process.

### 3.1 The Byzantine Tolerant Classifier

We present a classifier procedure that satisfies (B1)-(B3), shown in Fig. 4. It is inspired by the asynchronous classifier procedure given in [19] for the crash failure model. In the classifier procedure, each process stores a set of values received from other processes. We say a process writes a value to at least  $n - f$  processes if it sends a “write” message containing the value to all processes and waits for  $n - f$  processes to send acknowledgement back. We say a process reads from at least  $n - f$  processes if it sends a “read” message to all and waits for at least  $n - f$  processes to send their current values back. We say a process performs a write-read step if it writes its value to at least  $n - f$  processes and reads their values.

In the asynchronous classifier procedure for the crash failure model [19], to divide a group into a slave subgroup and a master subgroup, each process in the group first writes its value to at least  $n - f$  processes and then reads from at least  $n - f$  processes. After that, each process checks whether the union of all values obtained has size greater than the threshold parameter  $k$  or not. If true, it is classified as a master process, otherwise, it is classified as a slave process. Slave processes keep their values the same. To guarantee the value of each slave process is  $\leq$  the value of each master process, each master process performs a write-read step to write the values obtained at the read step to at least  $n - f$  processes and read the values from them. Then it updates its value to be the union of all values read. The second read step guarantees the size of the union of values of slave processes is  $< k$ , since the last slave process which completes the write step must have read all values of slave processes.

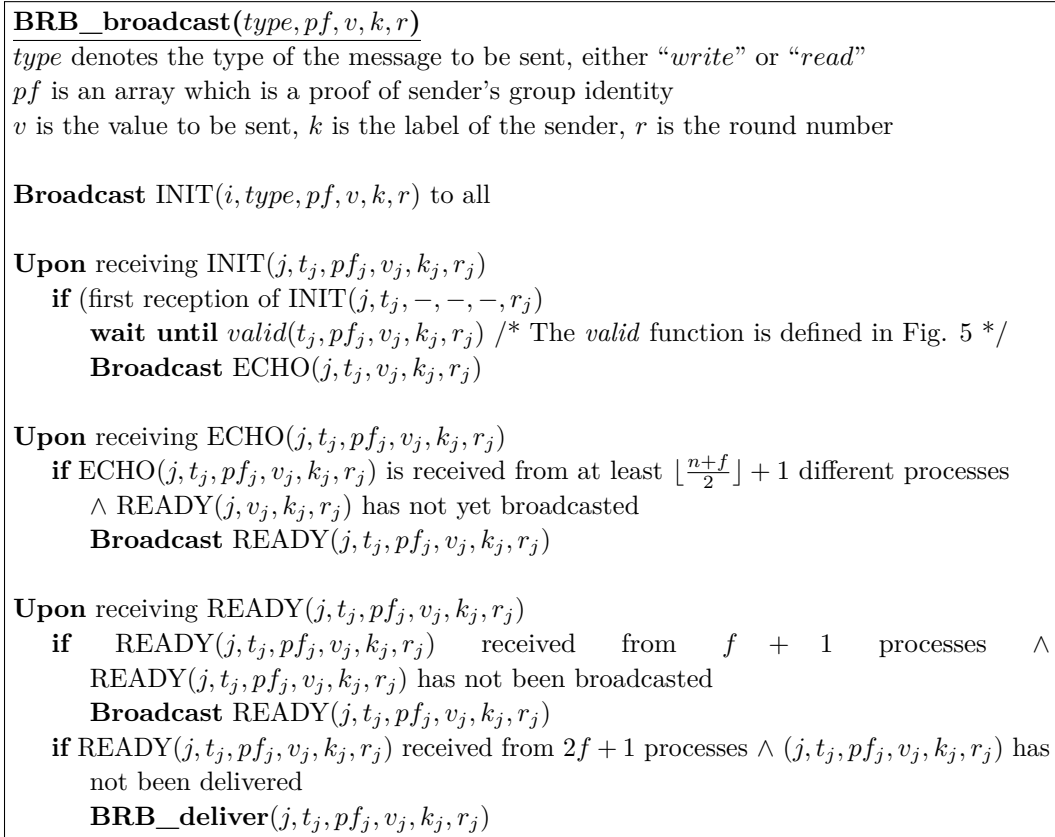
Constructing such a classifier procedure in presence of Byzantine processes is much more difficult. In order to adapt the above procedure to work in Byzantine setting, we need to address the following challenges. First, in the write step or read step, when a process waits for at least  $n - f$  different processes to send their values back, a Byzantine process can send arbitrary values. Second, simply ensuring that the values of a slave process is a subset of values of each master process is not enough, since a Byzantine process can introduce some values unknown to a master process in the slave group. For example, even if we can guarantee that the current value of each slave process is less than the value of each master process, in a later round, a Byzantine process can send some new value to a slave process which is unknown to some master process. This is possible in an asynchronous systems since messages can be arbitrarily delayed. Third, ensuring that the union of all values in the slave group has size at most  $k$  is quite challenging. A simple second read step does not work any more since the last process which completes the write step might be a Byzantine process.

To prevent the first problem, in the Byzantine classifier procedure, when a process wants to perform a write step or read step, it applies the reliable broadcast primitive to broadcast its value. When a process waits for values from at least  $n - f$  processes, it only accepts a value if the value is a subset of the values reliably delivered by this process. By property of reliable broadcast, this ensures that each accepted value must be reliably broadcast by some process, which prevents Byzantine processes from introducing arbitrary values.

To tackle the second and third problem, the key idea is to restrict the values that a Byzantine process, which claims itself to be a slave process, can successfully reliable broadcast in later rounds. To achieve that, first we require that that a slave process can only reliable broadcast the value that it has reliably broadcast in the previous round. This prevents Byzantine processes from introducing arbitrary new values into a slave group. Second, we require each process which claims itself as a slave process to prove that it is indeed classified as a slave at the previous round when it tries to reliable broadcast a value at the current round by presenting the set of values it used to do classification. To enforce the above two requirements, we add a validity condition when a process echoes a message in the reliable



broadcast primitive. However, this is not enough, since the value of a Byzantine slave process might not be known to a master process if the value of the Byzantine process is arbitrarily delayed. To ensure that the value a Byzantine process reliably broadcast is read by each correct master process, we force a Byzantine process who wants to be able to reliable broadcast a value in the slave group at next round to actually write its value to at least  $\lfloor \frac{n+f}{2} \rfloor + 1 - f$  correct processes, i.e., at least  $\lfloor \frac{n+f}{2} \rfloor + 1 - f$  correct processes must have received the value of a Byzantine process before each correct master process tries to read from at least  $n - 2f$  correct processes. These two sets of correct processes must have at least one correct process in common since  $f < \frac{n}{5}$ .



■ **Figure 3** Bounded Reliable Broadcast.

Each process which is classified as master is not required to prove its group identity but the value it tries to broadcast has to be a subset of safe value sets of correct processes. To ensure that the value of a slave process is less than the value of a master process, a master process needs to do a write-read step after it is classified as a master process.

**Bounded Reliable Broadcast.** Before explaining the Byzantine classifier procedure in detail, we modify the reliable broadcast primitive by adding a condition when a process echoes a broadcast message. This condition restricts the admissible values for each group. For completeness, the modified reliable broadcast procedure is shown in Fig. 3. When a process reliable broadcasts a value, it also includes the round number, its current label and a proof of its group identity. The proof is an array of size  $n$  denoting the values read by the sender at previous round, which will be explained in detail when we present the classifier

procedure. When a process  $i$  receives a broadcast message from process  $j$ , it waits for the validity condition to hold and then echoes the message. We say a process `BRB_broadcast` a message if it executes `BRB_broadcast` procedure with the message. We say a process `BRB_delivers` a message if it executes `BRB_deliver` with this message.

**Groups and Admissible Values.** In our algorithm, each process  $i$  has a label  $l_i$ , which serves as the threshold when it invokes the classifier procedure. The notion of group defined as below is based on labels of processes.

► **Definition 1 (group).** *A group is a set of processes which have the same label. The label of a group is the label of the processes in this group. The label of a group is also the threshold value processes in this group use to do classification.*

We also use label to indicate a group. A process is in group  $k$  if its message is associated with label  $k$ . Initially all processes are within the same group with label  $k_0 = n - \frac{f}{2}$ . The label of each process is updated at each round based on the classification result. For group  $k$  at round  $r$ , let  $s(k, r) = k - \frac{f}{2^{r+1}}$  and  $m(k, r) = k + \frac{f}{2^{r+1}}$ . We introduce the notion of admissible values for a group, which is the set of values that processes in the group can ever have.

► **Definition 2 (admissible values for a group).** *The admissible values for a group  $G$  with label  $k$  is the set of values that can be reliably delivered with label  $k$  if they are reliably broadcast by some process (possibly Byzantine) with label  $k$ .*

In our classifier, each process in group  $k$  updates its value set to a subset of the values which are reliably delivered with label  $k$ . Thus, the value set of each process in group  $k$  must be a subset of the admissible values for group  $k$ .

### 3.2 The Classifier Procedure

The classifier procedure for process  $i \in [n]$ , shown in Fig. 4, has three input parameters:  $V$  is the current value set of process  $i$ ,  $k$  is the threshold value used to do the classification, which is also the current label of process  $i$ , and  $r$  is the round number.

In lines 1-2, process  $i$  writes its current value set to at least  $n - f$  processes by using the `BRB_broadcast` procedure to send a “write” message. If process  $i$  is classified as a slave at the previous round, it needs to include the array of values it read from at least  $n - f$  processes at previous round as a proof of its group identity. This proof is used by every other process in the `valid` function to decide whether to echo the “write” message or not. When process  $i$  `BRB_delivers` a “write” message with label  $k$  at round  $r$ , it includes the value in it into its safe value set for group  $m(k, r)$ . The safe value set is used to restrict the set of values that can be delivered in the master group  $m(k, r)$ . Due to this step, we can see that the admissible values in the master subgroup must be a subset of the admissible values at the current group. Process  $i$  also includes the value contained in the “write” message into  $ACV_i^r[k]$ , which stores the set of values reliably delivered with label  $k$  at round  $r$ .

From line 3 to line 4, process  $i$  reads values from at least  $n - f$  processes by using the `BRB_broadcast` procedure to send a “read” message to all. In the `valid` function, each process  $j$  echos a “read” message from process  $i$  only if it has `BRB_delivered` the “write” message from process  $i$  sent at line 2. This step is used to ensure that for any process, possibly Byzantine, to read from other processes, it must have written its value to at least  $\lfloor \frac{n+f}{2} \rfloor + 1 - f$  correct processes, otherwise it cannot have enough processes echo its “read” message in the `BRB_broadcast`. When process  $i$  `BRB_delivers` a “read” message with label  $k$  from process  $j$  at round  $r$ , it records the set of values it has reliably delivered with

```

Classifier( $V, k, r$ ) for  $p_i$ :
 $V$ : input value set     $k$ : threshold value     $r$ : round number
/* Each process  $i \in [n]$  keeps track of the following variables */
Array  $LB_i^r$ .  $LB_i^r[j]$  denotes the label of process  $j$  sent along its values at round  $r$ 
Map  $S_i$ .  $S_i[k]$  denotes a safe value set for group  $k$ 
Map  $ACV_i^r$ .  $ACV_i^r[k]$  denotes the set of values accepted with label  $k$ , initially  $ACV_i^r[k] := \emptyset$ 
Map  $RV_i^r$ .  $RV_i^r[j]$  denote the values process  $i$  read from process  $j$  at round  $r$  at line 4
Map  $RT_i^r$ .  $RT_i^r[j]$  denote the values process  $j$  read from process  $i$  at round  $r$ .

/* write step*/
1: if  $isSlave(i, k, r)$  then  $pf := RV_i^{r-1}$  else  $pf := \emptyset$ 
2: BRB_broadcast("write",  $pf, V, k, r$ ), wait for  $wack(-, r)$  from  $n - f$  different processes

/* read step*/
3: BRB_broadcast("read",  $-, -, k, r$ ), wait for  $n - f$   $rack(R_j, r)$  s.t.  $R_j \subseteq ACV_i^r[k]$  from  $p_j$ 
4: Set  $RV_i^r[j] := R_j$  if  $R_j \subseteq ACV_i^r[k]$ , otherwise  $RV_i^r[j] := \emptyset$ 

/* Classification */
5: Let  $T := \bigcup_{j=1}^n RV_i^r[j]$ 
6: if  $|T| > k$ 
    /* write-read step */
7:   Send  $master(T, k, r)$  to all, wait for  $n - f$   $mack(R_j, r)$  from  $p_j$  s.t.  $R_j \subseteq ACV_i^r[k]$ 
8:   Define  $T' := \cup\{R_j \mid R_j \subseteq ACV_i^r[k], j \in [n]\}$ 
9:   return ( $T', master$ )
10: else
11:   return ( $V, slave$ )

Upon BRB_Deliver( $j, type, -, v, k, r$ )
  if  $type = "write"$ 
     $S_i[m(k, r)] := S_i[m(k, r)] \cup v$  /* Construct safe value set for group  $m(k, r)$  */
     $ACV_i^r[k] := ACV_i^r[k] \cup v$ 
     $LB_i^r[j] := k$  /* Record the label of a process at round  $r$  */
    Send message  $wack(-, r)$  to  $p_j$ 
  elif  $type = "read"$ 
     $RT_i^r[j] := ACV_i^r[k]$ 
    Send message  $rack(ACV_i^r[k], r)$  to  $p_j$ 

Upon receiving  $master(T, k, r)$  from  $p_j$ 
  wait until  $T \subseteq ACV_i^r[k]$ 
  Send message  $mack(ACV_i^r[k], r)$  to  $p_j$ 

```

■ **Figure 4** The Byzantine Tolerant Classifier Procedure.

## 4:10 Byzantine Lattice Agreement in Asynchronous Systems

label  $k$  in  $RT_i^r[j]$ . Then process  $i$  sends back a *rack* message along with the set of reliably delivered values with label  $k$  at round  $r$  to process  $j$ . At line 3, after the “*read*” message is sent, process  $i$  has to wait for valid *rack* message from  $n - f$  processes. A *rack* message is valid if the value set contained in it is a subset of  $ACV_i^r[k]$ , which is the set of values reliably delivered with label  $k$  at round  $r$ . Consider a  $rack(R_j, r)$  message from a correct process  $j$ . Since  $j$  is correct, each value in  $R_j$  must have been reliably delivered by process  $j$ . By property of reliable broadcast, each value in  $R_j$  will eventually be reliably delivered by process  $i$ , thus  $R_j \subseteq ACV_i^r[k]$ . Thus, eventually process  $i$  can obtain  $n - f$  valid *rack* message. To implement line 3, we need a concurrent thread to check the wait condition whenever a new message is reliably delivered and added into  $ACV_i^r[k]$ . At line 4, process  $i$  records the set of valid  $R_j$ 's obtained at line 3 into array  $RV_i^r$ . So, this array stores the values reliably delivered with label  $k$  that process  $i$  read from all processes. This array is used to do classification in line 5-11 and also used as the proof of group identity of process  $i$  when it writes at next round.

Line 5-11 is the classification step. Process  $i$  is classified as a master process if the size of the union of valid values obtained in the read step is greater than its label  $k$ , otherwise, it is classified as a slave process. If it is classified as a slave process, it returns its input value set. If it is classified as a master process, process  $i$  performs a write-read step by sending a *master* message which includes the set of values it uses to do classification to all and wait for  $n - f$  valid *mack* message back at line 7. Similar to line 3, a *mack* message is valid if each value contained in it has been reliably delivered with correct label. When a process receives a *master* message with value set  $T$  and label  $k$  at round  $r$ , it first waits until all values in  $T$  are reliably delivered. Then it sends back a *mack* message along with the set of values reliably delivered with label  $k$  at round  $r$ . The waiting is used to ensure that each value in  $T$  is valid, i.e., be reliably delivered, because a Byzantine process can send arbitrary values in its *master* message at line 7. By a similar reasoning as line 3, process  $i$  will eventually obtain valid *mack* message from at least  $n - f$  different processes. After the write-read step, at line 8, process  $i$  updates its value set to be the union of values obtained at line 7.

```

function valid( $j, type, pf, v, k, r$ ) for process  $i$ :
  if ( $type = \text{“write”} \wedge \neg isSlave(j, k, r) \wedge v \subseteq S_i[k]$ )
     $\vee (type = \text{“write”} \wedge isSlave(j, k, r) \wedge BRB\_deliver(j, \text{“write”}, -, v, LB_i^{r-1}[j], r - 1)$ 
       $\wedge pf[i] = RT_i^{r-1}[j] \wedge |\bigcup_{j=1}^n pf[j]| \leq LB_i^{r-1}[j])$ 
     $\vee (type = \text{“read”} \wedge BRB\_deliver(j, \text{“write”}, -, -, k, r))$ 
    return True
  else
    return False

function isSlave( $j, k, r$ ) for process  $i$ :
  if  $k = LB_i^{r-1}[j] - \frac{f}{2^r}$ 
    return True
  else
    return False

```

■ **Figure 5** The *valid* Function.

The *valid* function is defined in Fig. 5. In the this function, we first consider the “*write*” messages. If the message has been sent by a process that claims to be a master, then it is considered valid if the value  $v$  in this message is contained in the safe value set  $S_i[k]$ . If

the message has been sent by a process that claims to be a slave, then process  $i$  checks (1) whether process  $i$  has BRB\_delivered the “write” message containing the same value at the previous round, (2) whether the  $i^{\text{th}}$  entry in  $pf$  array matches the value process  $j$  read from  $i$  in the previous round, and (3) whether the the number of values contained in the proof  $pf$  is at most  $k$ . The condition (1) ensures that a slave process sends the same value as the previous round since a correct slave process must keep its value same as in the previous round. The condition (2) ensures that the proof sent by the slave process uses values that it read at round  $r - 1$ . The condition (3) checks that the sender classified itself correctly.

If the message is a “read” with label  $k$  at round  $r$ , process  $i$  considers it as valid if it BRB\_delivered a “write” message with label  $k$  at round  $r$  from the sender. This is used to make sure that the sender (possibly Byzantine) must complete its write step in line 1-2 before trying to read at line 3-4.

The *isSlave* function invoked in the *valid* function simply checks whether the label of the sender matches the label update rule by comparing it with the label at previous round.

### 3.3 Proof of Correctness

We first define the notion of *committing* a message. Due to space limitation, we omit the proof of most lemmas. The notations used in our proof are listed in Table. 2.

► **Definition 3.** *We say a process **commits** a message if it reliably broadcasts the message and the message is reliably delivered. A process **commits** a message at time  $t$  if this message is reliably delivered by the first process at time  $t$ .*

■ **Table 2** Notations.

Variable	Definition
$G$	A group of processes at round $r$ with label $k$
$slave(G)$	The slave subgroup of $G$ , i.e., the processes with label $s(k, r)$ at round $r + 1$
$master(G)$	The master subgroup of $G$ , i.e., the processes with label $m(k, r)$ at round $r + 1$
$V_i^r$	The value set of process $i$ at the beginning of round $r$
$S_i^r$	The safe value map of process $i$ at the beginning of round $r$ $S_i^r[k]$ is the safe value set of process $i$ for group $k$ at the beginning of round $r$
$U_k^r$	The set of admissible values for group $k$ at round $r$ , i.e., the set of values that can be committed along with a “write” message at round $r$ with label $k$

By properties of reliable broadcast, we observe that each process (possibly Byzantine) can commit at most one “write” message and at most one “read” message at each round. Define  $s(k, r) = k - \frac{f}{2^{r+1}}$  and  $m(k, r) = k + \frac{f}{2^{r+1}}$ . The variables we use in the proof are shown in Table. 2. Consider the classification step in group  $k$  at round  $r$ . The following lemma shows that if a Byzantine process wants to commit a “write” message  $m$  at round  $r + 1$  with a slave label, then it must commit a “write” message  $m'$  which contains the same value as  $m$  and a “read” message at round  $r$  with label  $k$ . Also, it must commit its “read” message before its “write” message at round  $r$  with label  $k$ .

► **Lemma 4.** *Suppose that process  $i$  (possibly Byzantine) commits a write message  $(i, \text{“write”}, -, V_i, s(k, r), r + 1)$ . Then*

- 1) *The message  $(i, \text{“read”}, -, -, k, r)$  and the message  $(i, \text{“write”}, -, V_i, k, r)$  must be committed by process  $i$ .*

## 4:12 Byzantine Lattice Agreement in Asynchronous Systems

- 2) Let  $t$  denote the time that message  $(i, \text{“read”}, -, -, k, r)$  is committed. Then, the message  $(i, \text{“write”}, -, V_i, k, r)$  must have been reliably delivered by at least  $\lfloor \frac{n+f}{2} \rfloor + 1 - f$  correct processes before time  $t$ .

The following lemma shows that the classifier provides the properties we defined.

► **Lemma 5.** *Let  $G$  be a group at round  $r$  with label  $k$ . Let  $L$  and  $R$  be two nonnegative integers such that  $L < k \leq R$ . If  $L < |V_i^r| \leq R$  for each correct process  $i \in G$ , and  $|U_k^r| \leq R$ , then*

- (p1) For each correct  $i \in \text{master}(G)$ ,  $k < |V_i^{r+1}| \leq R$
- (p2) For each correct  $i \in \text{slave}(G)$ ,  $L < |V_i^{r+1}| \leq k$
- (p3)  $U_{s(k,r)}^{r+1} \subseteq U_k^r$
- (p4)  $U_{m(k,r)}^{r+1} \subseteq U_k^r$
- (p5)  $|U_{m(k,r)}^{r+1}| \leq R$
- (p6)  $|U_{s(k,r)}^{r+1}| \leq k$
- (p7) For each correct  $j \in \text{master}(G)$ ,  $U_{s(k,r)}^{r+1} \subseteq V_j^{r+1}$
- (p8) Each correct  $i \in \text{slave}(G)$  can commit its value set at round  $r+1$ , i.e.,  $V_i^{r+1} \subseteq U_{s(k,r)}^{r+1}$
- (p9) Each correct  $j \in \text{master}(G)$  can commit its value set at round  $r+1$ , i.e.,  $V_j^{r+1} \subseteq U_{m(k,r)}^{r+1}$
- (p10)  $|\cup \{V_i^{r+1} \mid i \in \text{slave}(G) \cap C\}| \leq k$       (p11)  $|\cup \{V_i^{r+1} \mid i \in \text{master}(G) \cap C\}| \leq R$

**Proof.**

- (p1)-(p5): Implied by how processes are classified as slave or master in the classifier.
- (p6): (Sketch) Let  $P$  denote the set of processes who can commit a write message at round  $r+1$  with label  $s(k,r)$ . Part 2) of Lemma 4 implies that the write message of each  $i \in P$  at round  $r$  must have been reliably delivered by at least  $\lfloor \frac{n+f}{2} \rfloor + 1 - f$  correct processes. Let  $l \in P$  be the last process s.t its write message at round  $r$  is reliably delivered by at least  $\lfloor \frac{n+f}{2} \rfloor + 1 - f$  correct processes. Process  $l$  must have read all the values written by processes in  $P$  at round  $r$  due to quorum intersection. Due to quorum intersection and the condition to which processes echo write messages, process  $l$  must have read all values in  $U_{s(k,r)}^{r+1}$  at round  $r$  and  $l$  is classified as slave at round  $r$ , which indicates that  $U_{s(k,r)}^{r+1}$ .
- (p7): (Sketch) Let  $P$  denote the set of processes who commit a “write” message at round  $r+1$  with label  $s(k,r)$ . Lemma 4 implies that the write message of each process in  $P$  must have been reliably delivered by at least  $\lfloor \frac{n+f}{2} \rfloor + 1 - f$  correct processes. The condition to which correct processes echo write messages implies that at round  $r+1$ , each process in  $P$  sends the same value as round  $r$  in its write message. Quorum intersection guarantees that each master process must have read the values of each process in  $P$  in its reading step at round  $r$ . Thus,  $U_{s(k,r)}^{r+1} \subseteq V_j^{r+1}$  for each  $j$ .
- (p8): Since process  $i$  is correct, at round  $r$ , it must read from at least  $n-2f$  correct processes. Let  $Q$  denote this set of correct processes. Then, at round  $r+1$ , each process in  $Q$  will echo  $i$ 's write message. Thus, there will be  $\geq n-2f$  echo messages. Since  $f < \frac{n}{5}$ , we have  $n-2f \geq \lfloor \frac{n+f}{2} \rfloor + 1$ . Hence, the write message of  $i$  will be eventually reliably delivered.
- (p9): (Sketch) Any value in  $V_j^{r+1}$  will eventually be reliably delivered by each correct process and be included into the safe value set of each correct process for the group with label  $m(k,r)$ .  $V_j^{r+1}$  will be reliably delivered by each correct process at round  $r+1$ .
- (p10)-(p11): (p10) is implied by (p8) and (p6). (p11) is implied by (p9) and (p5). ◀

The following lemma shows that the value set of a correct process is non-decreasing.

► **Lemma 6.** *For any correct process  $i$  and round  $r$ ,  $V_i^r \subseteq V_i^{r+1}$ .*

The following lemma is used later to show that processes in the same group at the end of the algorithm must have the same set of values.

► **Lemma 7.** *Let  $G$  be a group of processes at round  $r$  with label  $k$ . Then*

- (1) *for each correct process  $i \in G$ ,  $k - \frac{f}{2^r} \leq |V_i^r| \leq k + \frac{f}{2^r}$*
- (2)  *$|U_k^r| \leq k + \frac{f}{2^r}$*

**Proof.** By induction on round number  $r$  and apply (p1)-(p2) and (p5)-p(6) of Lemma 5. ◀

► **Lemma 8.** *Let  $i$  and  $j$  be two correct processes that are within the same group  $G$  with label  $k$  at the beginning of round  $\log f + 1$ . Then  $V_i^{\log f+1}$  and  $V_j^{\log f+1}$  are equal.*

**Proof (Sketch).** By applying Lemma 7 at round  $\log f$  within the parent group of  $G$ , we can show that  $k' < |V_i^{\log f+1}| \leq k' + 1$  and  $|\cup \{V_i^{\log f+1}, V_j^{\log f+1}\}| \leq k' + 1$ , where  $k'$  is the label of the parent group of  $G$ . Thus,  $V_i^{\log f+1} = V_j^{\log f+1}$ . ◀

► **Lemma 9 (Comparability).** *For any two correct process  $i$  and  $j$ ,  $y_i$  and  $y_j$  are comparable.*

**Proof.** If process  $i$  and  $j$  are in the same group at the beginning of round  $\log f + 1$ , then by Lemma 8,  $y_i = y_j$ . Otherwise, let  $G$  be the last group that both  $i$  and  $j$  belong to. Suppose  $G$  is a group with label  $k$  at round  $r$ . Suppose  $i \in \text{slave}(G)$  and  $j \in \text{master}(G)$  without loss of generality. Then,  $V_i^{\log f+1} \subseteq U_{s(k,r)}^{r+1} \subseteq V_j^{r+1} \subseteq V_j^{\log f+1}$ , by (p8), (p6) (p7) and (p5) of Lemma 5 and Lemma 6. ◀

► **Theorem 10.** *There is an  $O(\log f)$  rounds algorithm for the BLA problem in asynchronous systems which can tolerate  $f < \frac{n}{5}$  Byzantine failures, where  $n$  is the number of processes in the system. The algorithm takes  $O(n^2 \log f)$  messages.*

## 4 An $O(\log f)$ Rounds Algorithm for the Authenticated BLA Problem

In this section, we present an  $O(\log f)$  round algorithm for the BLA problem in authenticated (i.e., assuming digital signatures and public-key infrastructure) setting that can tolerate  $f < \frac{n}{3}$  Byzantine failures by modifying the Byzantine tolerant classifier procedure in previous section. The Byzantine classifier procedure in authenticated setting is shown in Fig. 6. The primary difference lies in what a process does when it reliably delivers some message and the validity condition for echoing a broadcast message. The basic idea is to let a process sign the *ack* message that it needs to send. Each process uses the set of signed *ack* messages as proof of its completion of a write step or read step. In this section, we use  $\langle x \rangle_i$  to denote a message  $x$  signed by process  $i$ , i.e.,  $\langle x \rangle_i = \langle x, \sigma \rangle$ , where  $\sigma$  is the signature produced by process  $i$  using its private signing key. We say a message is correctly signed by process  $i$  if the signature within the message is a correct signature produced by process  $i$ .

**The Authenticated Byzantine Tolerant Classifier.** The classifier in the authenticated setting is shown in Fig. 6. The primary difference between the classifier in previous section and the authenticated classifier is that in the authenticated classifier each process uses signed messages as proof of its group identity.

At lines 1-2, each process writes its current value set by using the **BRB\_broadcast** procedure to send a “write” message. If the process is a slave process, it also includes the set of at least  $n - f$  signed *rack* messages it received at the previous round as a proof that it is indeed classified as a slave. At line 2, each process waits for correctly signed *wack* message from at least  $n - f$  different processes. This set of signed *wack* message is used as the proof



#### 4:14 Byzantine Lattice Agreement in Asynchronous Systems

of its completion of the write step when this process tries to read from other processes. When a process BRB\_delivers a “write” message, it performs similar steps as the algorithm in previous section except that it sends a signed *wack* message back.

<p><b>Classifier</b>(<math>V, k, r</math>):  <math>V</math>: input value set    <math>k</math>: threshold value    <math>r</math>: round number  Each process <math>i \in [n]</math> keeps track of the same variables as the classifier in Fig. 4  Set <math>RV_i^r</math>, which stores the set of signed <i>rack</i> message in the read step of previous round</p> <ol style="list-style-type: none"> <li>1: <b>if</b> <i>isSlave</i>(<math>k, r</math>) <b>then</b> <math>pf := RV_i^{r-1}</math> <b>else</b> <math>pf := \emptyset</math></li> <li>2: BRB_broadcast(“write”, <math>pf, v, k, r</math>), wait for <math>n - f</math> valid <math>\langle wack(-, r) \rangle_j</math> from <math>p_j</math></li> <li>3: Let <math>W</math> denote the set of <math>\langle wack \rangle_j</math> delivered at line 2</li> <li>4: Send <i>read</i>(<math>W, k, r</math>) to all, wait for <math>n - f</math> valid <math>\langle rack(R_j, r) \rangle_j</math> s.t. <math>R_j \subseteq ACV_i^r[k]</math> from <math>p_j</math></li> <li>5: Set <math>RV_i^r := \{ \langle rack(R_j, r) \rangle_j \mid R_j \subseteq ACV_i^r[k] \}</math></li> <li>6: Let <math>T := \cup \{ R_j \mid R_j \subseteq ACV_i^r[k] \}</math></li> <li>7: <b>if</b> <math> T  &gt; k</math> /* Size of <math>T</math> is greater than the threshold */</li> <li>8:     Send <i>master</i>(<math>T, k, r</math>) to all, wait for <math>n - f</math> <i>mack</i>(<math>R_j, r</math>) s.t. <math>R_j \subseteq ACV_i^r[k]</math> from <math>p_j</math></li> <li>9:     Define <math>T' := \cup \{ R_j \mid R_j \subseteq ACV_i^r[k] \}</math></li> <li>10:    <b>return</b> (<math>T', master</math>)</li> <li>11: <b>else</b></li> <li>12:    <b>return</b> (<math>V, slave</math>)</li> </ol>
<p><b>Upon BRB_deliver</b>(<math>j, t, v, k, r</math>)  <b>if</b> <math>t = \text{“write”}</math>  <math>S_i[m(k, r)] := S_i[m(k, r)] \cup v</math>,    <math>ACV_i^r[k] := ACV_i^r[k] \cup v</math>  Send message <math>\langle wack(ACV_i^r[k], r) \rangle_i</math> to <math>p_j</math></p> <p><b>Upon receiving</b> <i>read</i>(<math>W, k, r</math>) from <math>p_j</math>  <b>if</b> <i>validSignature</i>(“read”, <math>j, W, r</math>)  Send message <math>\langle rack(ACV_i^r[k], r) \rangle_i</math> to <math>p_j</math></p> <p><b>Upon receiving</b> <i>master</i>(<math>T, k, r</math>) from <math>p_j</math>  <b>wait until</b> <math>T \subseteq ACV_i^r[k]</math>  Send message <i>mack</i>(<math>ACV_i^r[k], r</math>) to <math>p_j</math></p>

■ **Figure 6** *The Authenticated Byzantine Tolerant Classifier.*

At line 4-5, each process reads from at least  $n - f$  processes. Different from the classifier procedure in previous section, each process directly sends a read message along with the set of correctly signed *wack* messages obtained at line 2 to all (instead of using the BRB\_broadcast procedure). When a process receives a “read” message with label  $k$  for round  $r$ , it uses the *validSignature* function to check whether the “read” message contains correctly signed *wack* message for round  $r$  from at least  $n - f$  different processes. If so, it sends back to the sender a signed *rack* message along with the reliably delivered values with label  $k$  at round  $r$ . This ensures that if a process (possibly Byzantine) tries to read from correct processes, it must complete its write step first.



The classification step from line 6-12 is the same as the classification step of the algorithm in previous section. A master process performs a write-read step by sending a *master* message along the set of value obtained at line 6. Then it waits for  $n - f$  valid *rack* messages and updates its value set to be the set of values contained in these messages. When a process receives a *master* message, it performs the same steps as in the classifier in previous section.

The *valid* function is different from the one given in previous section. First, only “*write*” messages are reliably broadcast. Second, the proof is a set of signed *rack* messages instead of an array in previous section. To verify the proof, the *valid* function invokes the *validSignature* function to check whether the proof contains correctly signed *rack* message for previous round from at least  $n - f$  different processes.

```

function valid(j, type, pf, v, k, r):
  if (type = “write”  $\wedge$   $\neg$ isSlave(j, k, r)  $\wedge$   $v \subseteq S_i[k]$ )
     $\vee$  (type = “write”  $\wedge$  isSlave(j, k, r)  $\wedge$  BRB_deliver(j, “write”,  $-$ , v,  $LB_i^{r-1}[j]$ , r - 1)
       $\wedge$  validSignature(“write”, pf, r)  $\wedge$  pf contains at most k distinct values
    return True
  else
    return False

function validSignature(type, pf, r):
  if (type = “write”  $\wedge$  pf contains correctly signed rack( $-$ , r - 1) from  $n - f$  processes)
     $\vee$  (type = “read”  $\wedge$  pf contains correctly signed wack( $-$ , r) from  $n - f$  processes)
    return True
  else
    return False

```

■ **Figure 7** The *valid* Function.

For the proof of correctness, we just need to prove the classifier procedure satisfies the properties given Lemma 5 under the assumption that  $f < \frac{n}{3}$ .

► **Lemma 11.** *Properties (p1) – (p11) of Lemma 5 hold for the authenticated Byzantine tolerant classifier.*

► **Theorem 12.** *There is an  $O(\log f)$  rounds algorithm for the BLA problem in authenticated asynchronous systems which can tolerate  $f < \frac{n}{3}$  Byzantine failures, where  $n$  is the number of processes in the system. The algorithm takes  $O(n^2 \log f)$  messages.*

## 5 Conclusion

In this paper, we present an  $O(\log f)$  rounds algorithm for the Byzantine lattice agreement problem in asynchronous systems which can tolerate  $f < \frac{n}{5}$  Byzantine failures. We also give an  $O(\log f)$  rounds algorithm for the authenticated setting that can tolerate  $f < \frac{n}{3}$  Byzantine failures. One open problem left is to design an algorithm which has resilience of  $f < \frac{n}{3}$  and takes  $O(\log f)$  rounds.

## References

- 1 Hagit Attiya, Maurice Herlihy, and Ophir Rachman. Atomic snapshots using lattice agreement. *Distributed Computing*, 8(3):121–132, 1995.
- 2 Hagit Attiya and Ophir Rachman. Atomic snapshots in  $O(n \log n)$  operations. *SIAM Journal on Computing*, 27(2):319–340, 1998.
- 3 Martin Biely, Zarko Milosevic, Nuno Santos, and Andre Schiper. S-paxos: Offloading the leader for high throughput state machine replication. In *2012 IEEE 31st Symposium on Reliable Distributed Systems*, pages 111–120. IEEE, 2012.
- 4 Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.
- 5 Soma Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Inf. Comput.*, 105(1):132–158, 1993.
- 6 Soma Chaudhuri, Maurice Herlihy, Nancy A Lynch, and Mark R Tuttle. Tight bounds for  $k$ -set agreement. *Journal of the ACM (JACM)*, 47(5):912–943, 2000.
- 7 Giuseppe Antonio Di Luna, Emmanuelle Anceaume, Silvia Bonomi, and Leonardo Querzoni. Synchronous byzantine lattice agreement in  $O(\log f)$  rounds. *arXiv preprint arXiv:2001.02670*, 2020.
- 8 Giuseppe Antonio Di Luna, Emmanuelle Anceaume, and Leonardo Querzoni. Byzantine generalized lattice agreement. *arXiv preprint arXiv:1910.05768*, 2019.
- 9 Danny Dolev and H Raymond Strong. Authenticated algorithms for Byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.
- 10 Jose M Faleiro, Sriram Rajamani, Kaushik Rajan, G Ramalingam, and Kapil Vaswani. Generalized lattice agreement. In *Proceedings of the 2012 ACM symposium on Principles of distributed computing*, pages 125–134. ACM, 2012.
- 11 Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.
- 12 Maurice Herlihy, Sergio Rajsbaum, and Mark R Tuttle. Unifying synchronous and asynchronous message-passing models. In *Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 133–142, 1998.
- 13 Marios Mavronicolasa. A bound on the rounds to reach lattice agreement. <http://www.cs.ucy.ac.cy/mavronic/pdf/lattice.pdf>, 2018.
- 14 Thomas Nowak and Joel Rybicki. Byzantine approximate agreement on graphs. In *33rd International Symposium on Distributed Computing (DISC 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.
- 15 Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.
- 16 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*, pages 386–400. Springer, 2011.
- 17 Jan Skrzypczak, Florian Schintke, and Thorsten Schütt. Linearizable state machine replication of state-based crdts without logs. *arXiv preprint arXiv:1905.08733*, 2019.
- 18 Xiong Zheng and Vijay K Garg. Byzantine lattice agreement in synchronous systems. In *34th International Symposium on Distributed Computing (DISC 2020)*, 2020.
- 19 Xiong Zheng, Vijay K. Garg, and John Kaippallimalil. Linearizable Replicated State Machines With Lattice Agreement. In Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller, editors, *23rd International Conference on Principles of Distributed Systems (OPODIS 2019)*, volume 153 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 29:1–29:16, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
- 20 Xiong Zheng, Changyong Hu, and Vijay K Garg. Lattice agreement in message passing systems. In *32nd International Symposium on Distributed Computing (DISC 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

# Heterogeneous Paxos

Isaac Sheff 

Max Planck Institute for Software Systems, Saarland Informatics Campus, Saarbrücken, Germany

<https://IsaacSheff.com>

[isheff@mpi-sws.org](mailto:isheff@mpi-sws.org)

Xinwen Wang 

Cornell University, Ithaca, NY, USA

<https://www.cs.cornell.edu/~xinwen/>

[xinwen@cs.cornell.edu](mailto:xinwen@cs.cornell.edu)

Robbert van Renesse 

Cornell University, Ithaca, NY, USA

<https://www.cs.cornell.edu/home/rvr/>

[rvr@cs.cornell.edu](mailto:rvr@cs.cornell.edu)

Andrew C. Myers 

Cornell University, Ithaca, NY, USA

<https://www.cs.cornell.edu/andru/>

[andru@cs.cornell.edu](mailto:andru@cs.cornell.edu)

---

## Abstract

In distributed systems, a group of *learners* achieve *consensus* when, by observing the output of some *acceptors*, they all arrive at the same value. Consensus is crucial for ordering transactions in failure-tolerant systems. Traditional consensus algorithms are homogeneous in three ways:

- all learners are treated equally,
- all acceptors are treated equally, and
- all failures are treated equally.

These assumptions, however, are unsuitable for cross-domain applications, including blockchains, where not all acceptors are equally trustworthy, and not all learners have the same assumptions and priorities. We present the first consensus algorithm to be heterogeneous in all three respects. Learners set their own mixed failure tolerances over differently trusted sets of acceptors. We express these assumptions in a novel *Learner Graph*, and demonstrate sufficient conditions for consensus.

We present *Heterogeneous Paxos*, an extension of Byzantine Paxos. Heterogeneous Paxos achieves consensus for any viable Learner Graph in best-case three message sends, which is optimal. We present a proof-of-concept implementation and demonstrate how tailoring for heterogeneous scenarios can save resources and reduce latency.

**2012 ACM Subject Classification** Computer systems organization → Redundancy; Computer systems organization → Availability; Computer systems organization → Reliability; Computer systems organization → Peer-to-peer architectures; Theory of computation → Distributed algorithms; Information systems → Remote replication

**Keywords and phrases** Consensus, Trust, Heterogeneous Trust

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2020.5

**Related Version** Technical Report at <https://arxiv.org/abs/2011.08253> [47].

**Supplementary Material** Implementation source: <https://github.com/isheff/charlotte-public>

## 1 Introduction

The rise of blockchain systems has renewed interest in the classic problem of consensus, but traditional consensus protocols are not designed for the highly decentralized, heterogeneous environment of blockchains. In a Consensus protocol, processes called *learners* try to decide on the same value, based on the outputs of some set of processes called *acceptors*, some of



© Isaac Sheff, Xinwen Wang, Robbert van Renesse, and Andrew C. Myers;  
licensed under Creative Commons License CC-BY

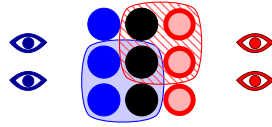
24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 5; pp. 5:1–5:17

Leibniz International Proceedings in Informatics



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



■ **Figure 1** Illustration of the scenario in § 1.1. Blue learners are drawn as blue eyes, red learners as red, outlined eyes. Blue acceptors are drawn as blue circles, red acceptors as red, outlined circles, and third parties as black circles. The light solid blue region holds a quorum for the blue learners, and the striped red holds a quorum for the red learners.

whom may fail. (In our model, learners send no messages, and so they cannot fail.) Consensus is a vital part of any fault-tolerant system maintaining strongly consistent state, such as Datastores [14, 9], Blockchains [41, 20, 16], or indeed anything which orders transactions. Traditionally, consensus protocols have been *homogeneous* along three distinct dimensions:

- Homogeneous acceptors. Traditional systems tolerate some number  $f$  of failed acceptors, but acceptors are interchangeable. Prior work including “failure-prone sets” [38, 27] explores *heterogeneous* acceptors.
- Homogeneous failures. Systems are traditionally designed to tolerate either purely Byzantine or purely crash failures. There is no distinction between failure scenarios in which the same acceptors fail, but possibly in different ways. However, some projects have explored *heterogeneous*, or “mixed” failures [48, 13, 33].
- Homogeneous learners. All learners make the same assumptions, so system guarantees apply either to all learners, or to none. Systems with *heterogeneous* learners include Cobalt [36] and Stellar [39, 35, 21].

Blockchain systems can violate homogeneity on all three dimensions. Permissioned blockchain systems like Hyperledger [1], J.P. Morgan’s Quorum [2], and R3’s Corda [26] exist specifically to facilitate atomic transactions between mutually distrusting businesses. A crucial part of setting up any implementation has been settling on a set of equally trustworthy, failure-independent acceptors. These setups are complicated by the reality that different parties make different assumptions about whom to trust, and how.

Defining heterogeneous consensus poses challenges not covered by homogeneous definitions, particularly with respect to learners. How should learners express their failure tolerances? When different learners expect different possible failures, when do they need to agree? If a learner’s failure assumptions are wrong, does it have any guarantees? No failure models developed for one or two dimensions of heterogeneity easily compose to describe all three.

Failure models developed for one or two dimensions of heterogeneity do not easily compose to describe all three, but our new trust model, the *Learner Graph* (§ 3), can express the precise trust assumptions of learners in terms of diverse acceptors and failures. Compared to trying to find a homogeneous setup agreeable to all learners, finding a learner graph for which consensus is possible is strictly more permissive. In fact, the learner graph is substantially more expressive than the models used in prior heterogeneous learner consensus work, including Stellar’s *slices* [39] or Cobalt’s *essential subsets* [36]. Building on our learner graph, we present the first fully *heterogeneous* consensus protocol. It generalizes Paxos to be heterogeneous along all three dimensions.

Heterogeneity allows acceptors to tailor a consensus protocol for the specific requirements of learners, rather than trying to force every learner to agree whenever any pair demand to agree. This increased flexibility can save time and resources, or even make consensus possible where it was not before, as we now show with an example.

## 1.1 Example

Suppose organizations **Blue Org** and **Red Org** want to agree on a value, such as the order of transactions involving both of their databases or blockchains. The people at **Blue Org** are *blue learners*: they want to decide on a value subject to *their* failure assumptions. Likewise, the people at **Red Org** are *red learners* with their own assumptions. While neither organization's learners believe their own organization's acceptors (machines) are Byzantine, they do not trust the other organization's acceptors at all. To help achieve consensus, they enlist three trustworthy third-party acceptors. Figure 1 illustrates this situation.

All learners want to agree so long as there are no Byzantine failures. However, no learner is willing to lose liveness (never decide on a value) if only one of its own acceptors has crashed, one third-party acceptor is Byzantine, and all the other organization's learners are Byzantine. Furthermore, learners within the same organization expect *never* to disagree, so long as none of their own organization's acceptors are Byzantine.

Unfortunately, existing protocols cannot satisfy these learners. Stellar [39], for instance, has one of the most expressive heterogeneous models available, but it cannot express heterogeneous failures. It cannot express *blue* and *red* learners' desire to terminate if a third-party acceptor crashes, but not necessarily agree a third-party acceptor is Byzantine. Our work enables a heterogeneous consensus protocol that satisfies all learners.

## 1.2 Heterogeneous Paxos

Heterogeneous Paxos, our novel generalization of Byzantine Paxos achieves consensus in a fully heterogeneous setting (§ 5), with precisely defined conditions under which learners are guaranteed safety and liveness. Heterogeneous Paxos inherits Paxos' optimal 3-message-send best-case latency, making it especially good for latency-sensitive applications with geodistributed acceptors, including blockchains. We have implemented this protocol and used it to construct several permissioned blockchains [21]. We demonstrate the savings in latency and resources that arise from tailoring consensus to specific learners' constraints.

## 1.3 Contributions

- The **Learner Graph** offers a general way to express heterogeneous trust assumptions in all three dimensions (§ 3).
- We **formally generalize the traditional consensus properties** (Validity, Agreement, and Termination) for the fully heterogeneous setting (§ 4).
- **Heterogeneous Paxos** is the first consensus protocol with heterogeneous learners, heterogeneous acceptors, and heterogeneous failures (§ 5). It also inherits Paxos' optimal 3-message-send best-case latency.
- **Experimental results** from our implementation of Heterogeneous Paxos demonstrate its use to construct permissioned blockchains with previously unobtainable security and performance properties (§ 6).

## 2 System Model

We consider a *closed-world* (or *permissioned*) system consisting of a fixed set of *acceptors*, a fixed set of *proposers*, and a fixed set of *learners*. Proposers and acceptors can send messages to other acceptors and learners. Some predetermined, but unknown set of acceptors are *faulty* (we assume a non-adaptive adversary). Faults include crash failures, which are not *live* (they can stop at any time without detection), and Byzantine failures, which are neither *live* nor *safe* (they can behave arbitrarily).

► **Definition 1 (Live).** *A live acceptor eventually sends every message required by the protocol.*

► **Definition 2 (Safe).** *A safe acceptor will not send messages unless they are required by the protocol, and will send messages only in the order specified by the protocol.*

Learners set the conditions under which they expect to agree. They want to decide values, and to be guaranteed agreement under certain conditions. While learners can make bad assumptions, since they do not send messages, they cannot misbehave, and so there are no “faulty learners.”

**Network.** Network communication is point-to-point and reliable: if a live acceptor sends a message to another live acceptor, or to a learner, the message arrives. We adopt a slight weakening of *partial synchrony* [18]: after some unknown global stabilization time (GST), all messages between live acceptors arrive within some unknown latency bound  $\Delta$ . In Heterogeneous Paxos, live acceptors send all messages to all acceptors and learners, but Byzantine acceptors may equivocate, sending messages to different recipients in different orders, with unbounded delays. We assume that messages carry effectively unbreakable cryptographic signatures, and that acceptors are identified by public keys. We also assume messages can **reference** other messages *by collision-resistant hash*: if one message contains a hash of another, it uniquely identifies the message it is referencing [42].

**Consensus.** The purpose of consensus is for each learner to decide on exactly one value, and for all learners to decide on the same value. Here, *execution* refers to a specific instance of consensus: the actions of a specific set of acceptors during some time frame. A *protocol* refers to the instructions that safe acceptors follow during an execution.

An execution of consensus begins when *proposers propose* candidate values, in the form of a message received by a correct acceptor. (No consensus can make guarantees about proposed values only known to crashed or Byzantine acceptors.) Proposers might be clients sending requests into the system. We make no assumptions about proposer correctness for safety properties, but to guarantee liveness, we will assume that acceptors can act as proposers as well (i.e. proposers are a superset of acceptors). After receiving some messages from acceptors, each learner eventually *decides* on a single value.

Traditionally, consensus requires three properties [19]:

- *Validity*: if a learner decides  $p$ , then  $p$  was proposed.<sup>1</sup>
- *Agreement*: if learner  $a$  decides value  $v$ , and learner  $b$  decides value  $v'$ , then  $v = v'$ .
- *Termination*: all learners eventually decide.

In § 4, we generalize these properties to account for heterogeneity.

### 3 The Learner Graph

We characterize learners’ failure assumptions with a novel construct called a *learner graph*. The learner graph is a general way to characterize trust assumptions for heterogeneous consensus. It can encompass most existing formulations, including Stellar’s “slices” [39] and Cobalt’s “essential sets” [36]. We discuss other formulations in § 7.

► **Definition 3 (Learner Graph).** *A learner graph is an undirected graph in which vertices are learners, each labeled with the conditions under which they must terminate (§ 4.3 formally defines termination). Each pair of learners is connected by an edge, labeled with the conditions under which those learners must agree (§ 4.2 formally defines agreement).*

<sup>1</sup> Correia, Neves, and Verissimo list several popular *validity* conditions. Ours corresponds to MCV2 [15]

### 3.1 Quorums

A *quorum* is a set of acceptors sufficient to make a learner decide: even if everything else has crashed [32], if a quorum are behaving correctly, a learner will eventually decide. In a learner graph, each learner  $a$  is labeled with a set of quorums  $Q_a$ . The learner requires termination precisely when at least one quorum are all live.

Within a specific execution, we assume some (unknown) set of pre-determined acceptors are actually *live*. We call this set  $\mathcal{L}$ .

### 3.2 Safe Sets

To characterize the conditions under which two learners want to agree, we need to express all possible failures they anticipate. Surprisingly, crash failures cannot cause disagreement: any disagreement that occurs when some acceptor has crashed could also occur if the same acceptor were correct, but very slow, and did not act until after the learner decided. Therefore, for agreement purposes, each tolerable failure scenario is characterized by a *safe set* (usually written  $s$ ), the set of acceptors who are *safe*, meaning they act only according to the protocol. Between any pair of learners  $a$  and  $b$  in the learner graph, we label the edge between them with a set of safe sets  $a-b$ : so long as one of the safe sets in  $a-b$  indeed comprises only safe acceptors, the learners demand agreement.

Within a specific execution, we assume some (unknown) set of pre-determined acceptors are actually *safe*. We call this set  $\mathcal{S}$ . We do not require it, but systems often assume that  $\mathcal{S} \subseteq \mathcal{L}$ , since a Byzantine acceptor [31] may choose not to send messages.

#### 3.2.1 Subset of Tolerable Failures

We generally assume that a subset of tolerable failures is always tolerated:

► **Assumption 4.** *Subset of failures properties:*

$$\begin{aligned} \forall \ell . q_a \in Q_a &\Rightarrow \ell \cup q_a \in Q_a \\ \forall x . s \in a-b &\Rightarrow x \cup s \in a-b \end{aligned}$$

One might imagine, for example, two learners who demand agreement if two acceptors fail, but not if only one acceptor fails. However, we have no guarantee on time: if two acceptors are indeed faulty, one might act normally for an indefinite time, so the system would act as though only one has failed, and we will have to guarantee agreement.

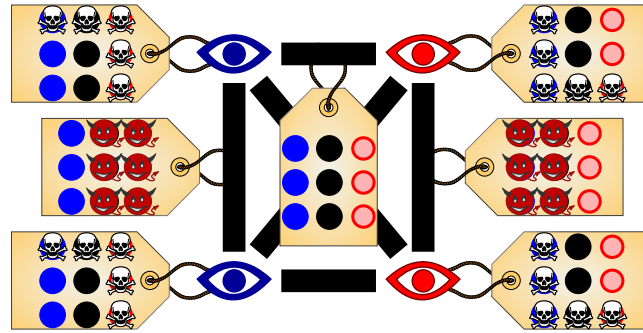
#### 3.2.2 Generalized Learner Graph Labels

It is possible to generalize the labels of learners and learner graph edges, and characterize quorums (conditions under which a learner must terminate) and safe sets (conditions under which pairs of learners must agree) as more detailed formal models (e.g., modeling network synchrony failures). All consensus failure models of which we are aware can be formalized using learner graphs with generalized labels. Heterogeneous Paxos works with any model of labels, so long as each label can be mapped (not necessarily uniquely) to a set of quorums for each learner, and a set of safe sets for each edge. For simplicity, in this work, we define labels as a set of quorums for each learner, and a set of safe sets for each edge.

### 3.3 Example

Consider our example from § 1.1 and Figure 1. All learners want to agree when all acceptors are safe. However, each learner demands termination (it must eventually decide on a value) even when one of its own acceptors has crashed, and one third part as well as all the





■ **Figure 2** Learner Graph from § 3.3: Learners are eyes, with darker **blue learners** on the left, and outlined **red learners** on the right. Edge labels display *one* safe set for which the learners want to agree (unsafe acceptors are marked with a devil). The center label represents all edges between **red** and **blue** learners. Learner labels display *one* quorum for which the learner wants to terminate (crashed acceptors are marked with a skull). In each label, **blue acceptors** are blue circles, **red acceptors** are red, outlined circles, and third-party acceptors are black circles.

other organization’s acceptors have failed as well. Furthermore, learners within the same organization expect *never* to disagree, so long as none of their own organization’s acceptors are Byzantine: neither organization tolerates the other, or third-party acceptors, creating internal disagreement. In Figure 2, we diagram the learner graph. For space reasons, we draw each label with only *one* quorum or *one* safe set.

### 3.4 Agreement is Transitive and Symmetric

Agreement (formally defined in § 4.2) is symmetric, so learner graphs are undirected ( $a-b = b-a$ ). Agreement is also transitive: if  $a$  agrees with  $b$  and  $b$  agrees with  $c$ , then  $a$  agrees with  $c$ . As a result,  $a$  and  $c$  must agree whenever both the conditions  $a-b$  and  $b-c$  are met. When learners’ requirements reflect this assumption, we call the resulting learner graph *condensed*. We describe how to condense a learner graph in § 3.5 of [47].

► **Definition 5** (Condensed Learner Graph (CLG)). *A learner graph  $G$  is condensed iff:  $\forall a, b, c. (a-b \cap b-c) \subseteq a-c$*

**Self-Edges.** A CLG describes when a learner  $a$  agrees with itself (i.e., if it decides twice, both decisions must have the same value):  $a-a$ .

► **Lemma 6** (Self-agreement). *A learner must agree with itself in order to agree with anyone:  $a-b \subseteq a-a$*

**Proof.** Follows from Definition 5, and the fact that the CLG is undirected (§ 3.4) ◀

### 3.5 Liveness Bounds from Safety

Given the conditions under which learners want to agree, we can derive a (sufficient) bound on the quorums they require to terminate. In other words, given labels for the edges in the learners graph, we can bound the labels for the vertices.

As we will cover in more detail in § 5.1, each of a learner’s quorums must intersect its neighbors quorums at a safe acceptor. As a result, we can construct a sufficient set of quorums for each learner in a CLG as follows: for each edge of the learner, each quorum includes a majority of acceptors from a each of the safety sets.



### 3.6 Safety Bounds from Liveness

Given the conditions under which learners want to terminate, we can derive a (necessary) bound on the safe sets they can require on each of their edges. As we will cover in more detail in § 5.1, each of a learner’s quorums must intersect its neighbors quorums at a safe acceptor. As a result, safe sets can be assembled for each edge in a CLG as follows: each set includes one acceptor from the intersection of each pair of quorums (one from each learner).

## 4 Heterogeneous Consensus

We now define our novel heterogeneous generalization of traditional consensus properties.

### 4.1 Validity

Intuitively, a consensus protocol shouldn’t allow learners to always decide some predetermined value. Validity is the same in heterogeneous and homogeneous settings.

► **Definition 7** (Heterogeneous Validity).

- *A consensus execution is valid if all values learners decide were proposed in that execution.*
- *A consensus protocol is valid if all possible executions are valid.*

### 4.2 Agreement

Our generalization of Agreement from the homogeneous setting to a heterogeneous one is the key insight that makes our conception of heterogeneous consensus possible. It generalizes not only the traditional homogeneous approach, but also the “intact nodes” concept from Stellar [39], and “linked nodes” from Cobalt [36].

► **Definition 8** (Entangled). *In an execution, two learners are entangled if their failure assumptions matched the failures that actually happen:  $Entangled(a, b) \triangleq S \in a-b$*

In the example (§ 1.1), if one third-party acceptor were Byzantine, the **blue learners** would be entangled with each other, and similarly with the **red learners**, but no **blue learners** would be entangled with **red learners**. It is possible for failures to divide the learners into separate groups, which may then decide different values even if they agree among themselves.

► **Definition 9** (Heterogeneous Agreement).

- *Within an execution, two learners have agreement if all decisions for either learner have the same value.*
- *A heterogeneous consensus protocol has agreement if, for all possible executions of that protocol, all entangled pairs of learners have agreement.*

In Heterogeneous Paxos, as in many other protocols, learners decide on a value whenever certain conditions are met for that value: learners can even decide multiple times. If there aren’t too many failures, a learner is guaranteed to decide the same value every time. Because learners send no messages, they cannot *fail*, but they can make incorrect assumptions. Within the context of an execution, entanglement neatly defines when a learner is *accurate*, meaning it cannot decide different values.

► **Definition 10** (Accurate Learner). *is entangled with itself:  $Accurate(a) \triangleq Entangled(a, a)$*

In the example (§ 1.1), if one third-party acceptor were Byzantine, then the **blue** and **red** learners would be accurate, but if a **blue acceptor** were also Byzantine, the **blue learners** would not be accurate (although the **red learners** would still be accurate).

```

1  acceptor_initial_state:
2  known_messages = {}
3  recently_received = {}
4
5  acceptor_on_receipt(m):
6  for r ∈ m.refs:
7  while r ∉ known_messages:
8  wait()
9  atomic:
10 if m ∉ known_messages:
11 forward m to all acceptors and learners
12 recently_received ∪= {m}
13 known_messages ∪= {m}
14 if m has type 1a:
15 z = new 1b(refs = recently_received)
16 recently_received = {}
17 on_receipt(z)
18 if m has type 1b and b(m) == maxx ∈ known_messages b(x)
19 for learner ∈ learners:
20 z = new 2a(refs = recently_received, lrn = learner)
21 if WellFormed(z):
22 recently_received = {}
23 on_receipt(z)

```

```

1  learner_initial_state:
2  known_messages = {}
3
4  learner_on_receipt(m):
5  for r ∈ m.refs:
6  while r ∉ known_messages:
7  wait()
8  known_messages ∪= {m}
9  for S ⊆ known_messages:
10 if Decisionself(S ∪ {m}):
11 decide(V(m))

```

■ **Figure 3** Pseudo-code for Acceptor (left) and Learner (right). § 5 defines message structure (§ 5.2), *WellFormed* (Assumption 26), *b()* (Definition 19), *V()* (Definition 20), and *Decision()* (Definition 21).

### 4.3 Termination

Termination has no well agreed-upon definition for the heterogeneous setting, as it does not generalize easily from the homogeneous one. A heterogeneous consensus protocol is specified in terms of the (possibly differing) conditions under which each learner is guaranteed termination (§ 3). For example, in our prior work on Heterogeneous Fast Consensus, we distinguish between “gurus,” learners with accurate failure assumptions, and “chumps,” who hold inaccurate assumptions [45]; Stellar calls them “intact” and “befouled” [39]. When discussing termination properties, we use the following terminology:

► **Definition 11** (Termination).

- *Within an execution, a learner has termination if it eventually decides.*
- *A heterogeneous consensus protocol has termination if, for all possible executions of that protocol, all learners with a safe and live quorum have termination.*

Protocols can only guarantee termination under specific network assumptions, and varying notions of “eventually” [19, 29, 40]. Following in the footsteps of Dwork et al. [18], Heterogeneous Paxos guarantees Validity and Agreement in a fully asynchronous network, and termination in a partially synchronous network (Assumption 31). Furthermore, as in all other consensus protocols, if there are too many acceptor failures, some learners may not terminate. Specifically, a learner will decide (terminate) if at least one of its quorums is live.

► **Definition 12** (Terminating Learner). *has a live, safe quorum: Terminating(a)  $\triangleq$   $\mathcal{L} \cup S \in Q_a$*

## 5 Heterogeneous Paxos

Heterogeneous Paxos is a consensus protocol (§ 2) based on Byzantine Paxos, Lamport’s Byzantine-fault-tolerant [31] variant of Paxos [28, 29] using a simulated leader [30]. This protocol is conceptually simpler than *Practical Byzantine Fault Tolerance* [10]. When all learners have the same failure assumptions, Heterogeneous Paxos is *exactly* Byzantine Paxos.

Byzantine Paxos was originally written as a sequence of changes from crash-tolerant Paxos [30, 28]. We were able to construct a complete version of Byzantine Paxos in such a way that we could describe Heterogeneous Paxos with only a few additions, highlighted in pale blue. To our knowledge, without the portions highlighted in pale blue this is also the most direct description of the Byzantine Paxos via Simulated Leader protocol in the literature. Figure 3 presents pseudocode for Heterogeneous Paxos acceptors and learners.

Informally, Heterogeneous Paxos proceeds as a series of (possibly overlapping) *phases* corresponding to three types of messages, traditionally called *1a*, *1b*, and *2a*:

- Proposers send *1a* messages, each carrying a value and unique *ballot number* (stage identifier), to acceptors.
- Acceptors send *1b* messages to each other to communicate that they've received a *1a* (line 15 of Figure 3).
- When an acceptor receives a *1b* message for the highest ballot number it has seen from a learner *a*'s *quorum* of acceptors, it sends a *2a* message labeled with *a* and that ballot number (line 20 of Figure 3). There is one exception (*WellFormed* in Figure 3): once a safe acceptor sends a *2a* message *m* for a learner *a*, it never sends a *2a* message with a different value for a learner *b*, unless:
  - It knows that a quorum of acceptors has seen *2a* messages with learner *a* and ballot number higher than *m*.
  - Or it has seen Byzantine behavior that proves *a* and *b* do not have to agree.
- A learner *a* *decides* when it receives *2a* messages with the same ballot number from one of its quorums of acceptors (line 11 on the right of Figure 3).

Proposers can restart the protocol at any time, with a new ballot number. Acceptor and Learner behavior in Heterogeneous Paxos is described in Figure 3. We now describe their sub-functions, including message construction (§ 5.2), *WellFormed* (Assumption 26), *b()* (Definition 19), *V()* (Definition 20), and *Decision()* (Definition 21).

**Key Insight.** Intuitively, Heterogeneous Paxos operates much like Byzantine Paxos, except that all acceptors execute the final phase separately for each learner. The shared phases allow learners to agree when possible, while the replicated final phase allows different learners to decide under different conditions. § 8 of [47] describes several heterogeneous consensus scenarios, as well as quorums for each learner.

## 5.1 Valid Learner Graph

Naturally, there are bounds on the learner graphs for which Heterogeneous Paxos can provide guarantees. Unlike traditional consensus, in a Heterogeneous Consensus learner graph, each learner *a* has its own set of quorums  $Q_a$ . These describe the learner's termination constraints: it may not terminate if all of its quorums contain a non-live acceptor (Definition 12). The notion of a *valid* learner graph generalizes the homogeneous assumption that every pair of quorums have a safe acceptor in their intersection.

Homogeneous Byzantine Paxos guarantees agreement (§ 4.2) when all pairs of quorums have  $\geq 1$  safe acceptor in their intersection. The heterogeneous case has a similar requirement:

► **Definition 13 (Valid Learner Graph).** *A learner graph is valid iff for each pair of learners  $a$  and  $b$ , whenever they must agree, all of their quorums feature at least one safe acceptor in their intersection:  $s \in a-b \wedge q_a \in Q_a \wedge q_b \in Q_b \Rightarrow q_a \cap q_b \cap s \neq \emptyset$*

## 5.2 Messaging

Acceptors send messages to each other. Live acceptors echo all messages sent and received to all other acceptors and learners, so if one live acceptor receives a message, all acceptors eventually receive it. When safe acceptors receive a message, they process and send resulting messages specified by the protocol atomically: they do not receive messages between sending results to other acceptors. Safe acceptors also receive any messages they send to themselves immediately: they receive no other messages between sending and receiving.

Each message  $x$  contains a cryptographic signature allowing anyone to identify the signer:

► **Definition 14** (Message Signer).  $Sig(x: message) \triangleq$  the acceptor or proposer that signed  $x$

We can define  $Sig()$  over sets of messages, to mean the set of signers of those messages:

► **Definition 15** (Message Set Signers).  $Sig(x: set) \triangleq \{ Sig(m) \mid m \in x \}$

Furthermore, each message  $x$  carries references to 0 or more other messages,  $x.refs$ . These references are by hash, ensuring both the absence of cycles in the reference graph and that it is possible to know exactly when one message references another [42]. In each message, safe acceptors reference each message they received since the last message they sent. Since all messages sent are sent to all acceptors, and safe acceptors receive messages sent to themselves immediately, each message a safe acceptor sends *transitively* references all messages it has ever sent or received. Safe acceptors delay receipt of any message until they have received all messages it references. This ensures they receive, for example, a  $1a$  for a given ballot before receiving any  $1b$ s for that ballot.

Each message has a unique ID and an identifiable type:  $1a$ ,  $1b$ , or  $2a$ . A  $2a$  message  $x$  has one type-specific field:  $x.lrn$  specifies a learner. A  $1a$  message  $y$  has two type-specific fields:  $y.value$  is a proposed value, and  $y.ballot$  is a natural number specific to this proposal.

We assume that each  $1a$  has a unique ballot number, which could be accomplished by including signature information in the least significant bits of the ballot number:

► **Assumption 16** (Unique ballot assumption).  $z:1a \wedge y:1a \wedge z.ballot = y.ballot \Rightarrow z = y$

## 5.3 Machinery

To describe Heterogeneous Paxos, we require some mathematical machinery.

**Transitive References.** We define  $Tran(x)$  to be the transitive closure of message  $x$ 's references. Intuitively, these are all the messages in the “causal past” of  $x$ .

► **Definition 17.**  $Tran(x) \triangleq \{x\} \cup \bigcup_{m \in x.refs} Tran(m)$

**Get1a:** It is useful to refer to the  $1a$  that started the ballot of a message: the highest ballot number  $1a$  in its transitive references.

► **Definition 18.**  $Get1a(x) \triangleq \underset{m:1a \in Tran(x)}{\operatorname{argmax}} m.ballot$

**Ballot Numbers.** The ballot number of a  $1a$  is part of the message, and the ballot number of anything else is the highest ballot number among the  $1a$ s it (transitively) references.

► **Definition 19.**  $b(x) \triangleq Get1a(x).ballot$

**Value.** The value of a  $1a$  is part of the message, and the value of anything else is the value of the highest ballot  $1a$  among the messages it (transitively) references.

► **Definition 20.**  $V(x) \triangleq Get1a(x).value$

**Decisions.** A learner decides when it has observed a set of  $2a$  messages with the same ballot, sent by a quorum of acceptors. We call such a set a *decision*:

► **Definition 21.**  $Decision_a(q_a) \triangleq Sig(q_a) \in Q_a \wedge \forall \{x, y\} \subseteq q_a. b(x) = b(y) \wedge x.lrn = a \wedge x : 2a$

Messages in a decision share a ballot (and therefore a value), so we extend our value function to include decisions:  $Decision_a(q_a) \Rightarrow V(q_a) = V(m) \mid m \in q_a$

Although decisions are not messages, applications might send decisions in other messages as a kind of “proof of consensus.” This is how the Heterogeneous Paxos integrity attestations work in our prototype blockchains (§ 6).

**Caught.** Some behavior can create proof that an acceptor is Byzantine. Unlike Byzantine Paxos, our acceptors and learners must adapt to Byzantine behavior. We say that an acceptor  $p$  is *Caught* in a message  $x$  if the transitive references of the messages include evidence such as two messages,  $m$  and  $m'$ , both signed by  $p$ , in which neither is featured in the other’s transitive references (safe acceptors transitively reference all prior messages).

► **Definition 22.**  $Caught(x) \triangleq \left\{ Sig(m) \mid \begin{array}{l} \{m, m'\} \subseteq Tran(x) \wedge Sig(m) = Sig(m') \\ \wedge m \notin Tran(m') \wedge m' \notin Tran(m) \end{array} \right\}$

**Connected.** When some acceptors are proved Byzantine, clearly some learners need not agree, meaning that  $\mathcal{S}$  isn’t in the edge between them in the CLG: at least one acceptor in each safe set in the edge is proven Byzantine. Homogeneous learners are always connected unless there are so many failures no consensus is required.

► **Definition 23.**  $Con_a(x) \triangleq \{ b \mid s \in a-b \in CLG \wedge s \cap Caught(x) = \emptyset \}$

It is clear that disconnected learners may not agree, and so each  $2a$  message  $x$  will have some implications only for learners still connected to its specified learner:  $Con_{x.lrn}(x)$ .

**Quorums in Messages.**  $2a$  messages reference *quorums of messages* with the same value and ballot. A  $2a$ ’s quorums are formed from fresh  $1b$  messages with the same ballot and value (we define *fresh* in Definition 28).

► **Definition 24.**  $q(x : 2a) \triangleq \{ m \mid m : 1b \wedge fresh_{x.lrn}(m) \wedge m \in Tran(x) \wedge b(m) = b(x) \}$

**Buried messages.** A  $2a$  message can become irrelevant if, after a time, an entire quorum of acceptors has seen  $2as$  with different values, the same learner, and higher ballot numbers. We call such a  $2a$  *buried* (in the context of some later message  $y$ ):

► **Definition 25.**

$Buried(x : 2a, y) \triangleq \left\{ Sig(m) \mid \begin{array}{l} m \in Tran(y) \wedge z : 2a \wedge \{x, z\} \subseteq Tran(m) \\ \wedge V(z) \neq V(x) \wedge b(z) > b(x) \wedge z.lrn = x.lrn \end{array} \right\} \in Q_{x.lrn}$

## 5:12 Heterogeneous Paxos

**Well-Formedness.** In addition to the basic message layout,  $2a$  and  $1b$  messages must be *well-formed*. No  $2a$  should have an invalid quorum upon creation, and no acceptor should create a  $2a$  unless it sent one of the  $1b$  messages in the  $2a$ . Similarly, no  $1b$  should reference any message with the same ballot number besides a  $1a$  (safe acceptors make  $1b$ s as soon as they receive a  $1a$ ). Acceptors and learners should ignore messages that are not well-formed.

► **Assumption 26** (Well-Formedness Assumption).

$$\begin{aligned} x : 1b \wedge y \in \text{Tran}(x) \wedge x \neq y \wedge y \neq \text{Get1a}(x) &\Rightarrow b(y) \neq b(x) \\ z : 2a \Rightarrow q(z) \in Q_{z.lrn} \wedge \text{Sig}(z) \in \text{Sig}(q(z)) \end{aligned}$$

**Connected  $2a$  messages.** Entangled learners must agree, but learners that are not connected are not entangled, so they need not agree. Intuitively, a  $1b$  message references a  $2a$  message to demonstrate that some learner may have decided some value. For learner  $a$ , it can be useful to find the set of  $2a$  messages from the same sender as a message  $x$  (and sent earlier) which are still unburied, and for learners connected to  $a$ . The  $1b$  cannot be used to make any new  $2a$  messages for learner  $a$  that have values different from these  $2a$  messages.

► **Definition 27.**  $\text{Con2as}_a(x) \triangleq \left\{ m \mid \begin{array}{l} m : 2a \wedge m \in \text{Tran}(x) \wedge \text{Sig}(m) = \text{Sig}(x) \\ \wedge \neg \text{Buried}(m, x) \wedge m.lrn \in \text{Con}_a(x) \end{array} \right\}$

**Fresh  $1b$  messages.** Acceptors send a  $1b$  message whenever they receive a  $1a$  message with a ballot number higher than they have yet seen. However, this does not mean that the  $1b$ 's value (which is the same as the  $1a$ 's) agrees with that of  $2a$  messages the acceptor has already sent. We call a  $1b$  message *fresh* (with respect to a learner) when its value agrees with that of unburied  $2a$  messages the acceptor has sent.

► **Definition 28.**  $\text{fresh}_a(x : 1b) \triangleq \forall m \in \text{Con2as}_a(x). V(x) = V(m)$

## 5.4 Ballots

Heterogeneous Paxos can be thought of as taking place in *stages* identified by natural numbers called ballots. § 5.6.3 of [47] describes one way to construct unique ballot numbers.

**Multiple Ballots.** Proposers construct new  $1a$  messages (with a value and a unique ballot number), and send them to all acceptors. Just like in Homogeneous Byzantine Consensus, it is possible for a ballot to *fail*: after some number of ballots, it may be the case that all messages have arrived, the protocol in Figure 3 doesn't require any acceptor to send any further messages, and yet no learner has decided. For this reason, it is necessary to start a new ballot when an old one is failing.

One way to handle this is to leave the responsibility at the proposers: if a proposer proposes a ballot, and learners don't decide for a while, then the proposer should propose again. Randomized exponential backoff can be used to allow clients to adapt to the unknown delay in a partially synchronous [18] network without flooding the system.

Another way is to have acceptors propose after a ballot has failed: when sufficiently many  $1b$  messages for a given ballot are collected, but none are fresh, an acceptor could send a new  $1a$ . There are subtleties to ensuring liveness, which we discuss in § 6.4.1 of [47].

## 5.5 Safety

Under our assumptions (§ 5.2 of [47]), Heterogeneous Paxos has the safety properties of Validity and Agreement (proofs in § 6.2 of [47] and § 6.3 of [47]):

► **Theorem 29** (Validity). *Heterogeneous Paxos is Valid (Definition 7):*

$$\text{Decision}_a(q_a) \Rightarrow \exists x : 1a. V(x) = V(q_a)$$

► **Theorem 30** (Agreement). *Heterogeneous Paxos has Agreement (Definition 9):*

$$\text{Entangled}(a, b) \wedge \text{Decision}_a(q_a) \wedge \text{Decision}_b(q_b) \Rightarrow V(q_a) = V(q_b)$$

## 5.6 Liveness

Heterogeneous Paxos, and indeed Byzantine Paxos, rely on a weak network assumption to guarantee termination. The assumption is complex precisely because it is weak; a simpler but stronger assumption, such as a partially synchronous network, would suffice.

► **Assumption 31** (Network Assumption). *To guarantee that a learner  $a$  decides, we assume that for some quorum  $q_a \in Q_a$  of safe and live acceptors:*

- *Eventually, there will be 13 consecutive periods of any duration, with no time in between, numbered 0 through 12, such that any message sent to  $a$  or an acceptor in  $q_a$  before one period begins is delivered before it ends.*
- *If an acceptor in  $q_a$  sends a message in between receiving two messages  $m$  and  $m'$  (and it receives no other messages in between), and  $m$  is delivered in some period  $n$ , then the message is sent in period  $n$ .*
- *No 1a message except  $x$ ,  $y$ , and  $z$  is delivered to any acceptor in  $q_a$  during any period.*
- *$x$  is delivered to an acceptor in  $q_a$  in period 0,  $y$  is delivered to an acceptor in  $q_a$  in period 4, and  $z$  is delivered to an acceptor in  $q_a$  in period 9.*
- *$V(y) = V(z)$  is the value of the highest ballot  $2a$  known to any acceptor in  $q_a$  at the end of period 3.*
- *$b(x)$  is greater than any ballot number of any message delivered to any acceptor in  $q_a$  before period 0, and  $b(x) < b(y) < b(z)$ .*

This assumption is *only necessary* for termination, not any safety property. We prove our termination theorem in § 6.4.1 of [47].

► **Theorem 32** (Termination). *If Assumption 31 holds for learner  $a$ , then  $a$  has Termination (Definition 11). Specifically, after period 12:  $\text{Terminating}(a) \Rightarrow \exists q_a. \text{Decision}_a(q_a)$  If Assumption 31 holds for all terminating learners, then Heterogeneous Paxos has Termination.*

A *partially synchronous* network is one in which, after some point in time, there exists some (possibly unknown) constant latency  $\Delta$  such that all sent messages arrive within  $\Delta$  [18]. We explain elsewhere how to add artificial message receipt delays to Heterogeneous Paxos in order to guarantee Assumption 31 in a partially synchronous network (§ 6.4.2 of [47]).

## 6 Implementation

Since Heterogeneous Paxos is designed for cross-domain applications where different parties have different trust assumptions, it is well-suited for blockchains. We constructed a variety of example blockchains using the Charlotte framework [46], which allows for pluggable integrity (consensus) mechanisms. Our servers are implemented in 1,704 lines of open-source Java. Charlotte uses 256-bit SHA3 hashes, P256 elliptic curve signatures, protobufs [43] for marshaling, and gRPC [24] for transmitting messages over TLS 1.3 channels.



To explore the performance of Heterogeneous Paxos, we created several blockchains with different CLGs (§ 3). The results (§ 9.3 of [47]) show that heterogeneous configurations save resources and latency compared with homogeneous configurations tolerating the same failures. For instance, in our example configuration § 1.1, a Homogeneous configuration tolerating similar failures would cost an extra 7 unnecessary acceptors, increasing latency overhead by 51% relative to Heterogeneous Paxos. *2a* messages include a quorum of 256-bit message hashes, so they expand linearly with quorum size, as does the cost of unmarshaling and verifying the signatures of the messages referenced. In all experiments, however, computational overhead was dominated by the theoretical minimum (simulated) geodistributed network latency.

## 7 Related Work

**Heterogeneous Acceptors and Failures.** Heterogeneous Paxos is based on Leslie Lamport’s Byzantine-fault-tolerant variant [30] of Paxos [28]. Byzantine Paxos supports heterogeneous acceptors because it uses quorums: not all acceptors need be of equal worth, but all quorums are. Although Lamport does not describe it explicitly, Byzantine Paxos can have heterogeneous, or *mixed* [48], failures, so long as quorum intersections have a safe acceptor and at least one quorum is safe and live.

Many papers have investigated hybrid failure models [48, 13, 7, 33] in which different consensus protocol acceptors can have different failure modes, including crash failures and Byzantine failures (heterogeneous failures). These papers typically investigate how many failures in each class can be tolerated. Other papers have looked at system models in which different acceptors may be more or less likely to fail [22, 38], or where failures are dependent (heterogeneous acceptors) [27, 17, 25].

Further generalizations are possible. Our Learner Graph uses only *safe* and *live* acceptors, but its labels might be generalized to support other failure types such as rational failures [3]. We have only considered learners that all make the same (weak) synchrony assumption, but others have studied learners with heterogeneous network assumptions [5, 37].

**Heterogeneous Learners.** Unlike ours, most related work conflates learners and acceptors. Early related work on “Consensus with Unknown Participants” [11, 23, 4] defines protocols in which each participant knows only a subset of other participants, inducing a “who-knows-whom” digraph; this work identifies properties of this graph that must hold to achieve consensus. Not every participant knows all participants, but trust assumptions are homogeneous: participants have the same beliefs about trustworthiness of other participants.

Our prior work describes [45] a heterogeneous failure model in which different participants may have different failure assumptions about other participants. We distinguished learners whose failure assumptions are accurate from those whose failure assumptions are inaccurate and we specified a heterogeneous consensus protocol in terms of the possibly different conditions under which each learner is guaranteed agreement. The paper constructs a heterogeneous consensus protocol that meets the requirements of all learners using lattice-based information flow to analyze and prove protocol properties.

Heterogeneous learners became of interest to blockchain implementations based on voting protocols where open membership was desirable. Ripple (XRP) [44] was the earliest blockchain to attempt support for heterogeneous learners. Originally, each learner had its own Unique Node List (UNL), the set of acceptors that it partially trusts and uses for making decisions. An acceptor in more UNLs is implicitly more influential. The protocol was updated because of correctness issues [12], and support for diverse UNLs was all but eliminated. Ripple has



proposed a protocol called Cobalt [36], in which each learner specifies a set of acceptors they partially trust, and it works if those sets intersect “enough.” Cobalt does not account for heterogeneous failures, and only limited acceptor heterogeneity.

The Stellar Consensus [39, 34, 35] blockchain protocol supports both heterogeneous learners and acceptors, although it does not distinguish the two; each learner specifies a set of “quorum slices.” Like Cobalt, Stellar does not account for heterogeneous failures. Neither Stellar nor Cobalt match Heterogeneous Paxos’ best-case latency. Heterogeneous Paxos inherits Byzantine Paxos’ 3-message-send best case latency, which is optimal for a consensus tolerating  $\lceil \frac{n}{3} \rceil - 1$  failures in the homogeneous Byzantine case or  $\lceil \frac{n}{2} \rceil - 1$  failures in the homogeneous crash case [6]. However, both Cobalt and Stellar are designed for an “open-world” model, where not all acceptors and learners are known in advance. We have not yet adapted Heterogeneous Paxos to an open-world setting.

The heterogeneous learner models of Cobalt and Stellar have been studied in detail by García-Pérez and Gotsman [21]. Cachin and Tackmann examine Stellar-style asymmetric trust models, including in shared-memory environments [8]. However, neither paper separates learners from acceptors, attempts to solve consensus, or considers heterogeneous failures; the Learner Graph is more general.

Like our work, *Flexible BFT* [37] distinguishes learners from acceptors and accounts for both heterogeneous learners and heterogeneous failures. It does not allow heterogeneous acceptors: they are interchangeable, and quorums are specified by size. Flexible BFT also has optimal best-case latency. It does not support crash failures, but introduces a new failure type called *alive-but-corrupt* for acceptors interested in violating safety but not liveness.

## 8 Conclusion

Heterogeneous Paxos is the first consensus protocol with heterogeneous acceptors, failures, and learners. It is based on the Learner Graph, a new and expressive way to capture learners’ diverse failure-tolerance assumptions. Heterogeneous consensus facilitates a more nuanced approach that can save time and resources, or even make previously unachievable consensus possible. Heterogeneous Paxos is proven correct against our new generalization of consensus for heterogeneous settings. This approach is well-suited to systems spanning heterogeneous trust domains; for example, we demonstrate working blockchains with heterogeneous trust.

Future work may expand learner graphs to represent even more types of failures. Heterogeneous Paxos may be extended to allow for changing configurations, or improved efficiency in terms of bandwidth and computational overhead. New protocols can also make use of our definition of heterogeneous consensus, perhaps adding new guarantees such as probabilistic termination in asynchronous networks.

---

## References

- 1 An introduction to Hyperledger, 2018.
- 2 Quorum whitepaper, 2018.
- 3 A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J. Martin, and C. Porth. BAR fault tolerance for cooperative services. In *SOSP*, pages 45–58, 2005.
- 4 E. A. Alchieri, A. N. Bessani, J. Silva Fraga, and F. Greve. Byzantine consensus with unknown participants. In *OPODIS*, pages 22–40, 2008.
- 5 E. Blum, J. Katz, and J. Loss. Synchronous consensus with optimal asynchronous fallback guarantees. In *Theory of Cryptography*, pages 131–150, 2019.
- 6 G. Bracha and S. Toueg. Resilient consensus protocols. In *PODC*, pages 12–26, 1983.

- 7 C. Cachin and M. Backes. Reliable broadcast in a computational hybrid model with byzantine faults, crashes, and recoveries. In *DSN*, 2003.
- 8 C. Cachin and B. Tackmann. Asymmetric distributed trust. In *OPODIS*, 2019.
- 9 B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, and H. Simitci et al. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *SOSP*, 2011.
- 10 M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *OSDI*, 1999.
- 11 D. Cavin, Y. Sasson, and A. Schiper. Consensus with unknown participants or fundamental self-organization. In *ADHOC-NOW*, 2004.
- 12 B. Chase and E. MacBrough. Analysis of the XRP ledger consensus protocol. *CoRR*, abs/1802.07242, 2018.
- 13 A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. Upright cluster services. In *SOSP*, pages 277–290, 2009.
- 14 J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, and P. Hochschild et al. Spanner: Google’s globally distributed database. *TOCS*, 31(3):8, 2013.
- 15 M. Correia, N. Neves, and P. Veríssimo. From consensus to atomic broadcast: Time-free byzantine-resistant protocols without signatures. *Comput. J.*, 49:82–96, January 2006.
- 16 K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. Kosba, A. Miller, P. Saxena, E. Shi, E. G. Sirer, D. Song, and R. Wattenhofer. On scaling decentralized blockchains. In *Financial Cryptography and Data Security*, 2016.
- 17 C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, and A. Tielmann. The disagreement power of an adversary. *Distributed Computing*, 24:137–147, November 2011.
- 18 C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988.
- 19 M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.
- 20 Ethereum Foundation. Ethereum white paper. Technical report, Ethereum Foundation, 2018.
- 21 Á. García-Pérez and A. Gotsman. Federated byzantine quorum systems. In *OPODIS*, pages 17:1–17:16, 2018.
- 22 D. K. Gifford. Weighted voting for replicated data. In *SOSP*, 1979.
- 23 F. Greve and S. Tixeuil. Knowledge connectivity vs. synchrony requirements for fault-tolerant agreement in unknown networks. In *DSN*, pages 82–91, 2007.
- 24 grpc: A high performance, open-source universal RPC framework. <https://grpc.io>, 2018.
- 25 R. Guerraoui and M. Vukolić. Refined quorum systems. In *PODC*, 2007.
- 26 M. Hearn and R. G. Brown. Corda: A distributed ledger. Technical report, r3, 2019.
- 27 F. Junqueira and K. Marzullo. Designing algorithms for dependent process failures. In *Workshop on Future Directions in Distributed Computing*, pages 24–28, 2003.
- 28 L. Lamport. The Part-time Parliament. *TOCS*, 16(2):133–169, May 1998.
- 29 L. Lamport. Paxos made simple. Technical report, Microsoft Research, December 2001.
- 30 L. Lamport. Byzantizing Paxos by refinement. In *DISC*, pages 211–224, 2011.
- 31 L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Trans. on Programming Languages and Systems*, 4(3):382–401, July 1982.
- 32 B. W. Lampson and H. E. Sturgis. Crash recovery in a distributed data storage system. Technical report, Xerox Palo Alto Research Center, Palo Alto, CA, 1979.
- 33 S. Liu, P. Viotti, C. Cachin, V. Quéma, and M. Vukolic. XFT: Practical fault tolerance beyond crashes. In *OSDI*, 2016.
- 34 M. Lokhava, G. Losa, D. Mazières, G. Hoare, N. P. E. Barry, E. Gafni, J. Jové, R. Malinowsky, and J. M. McCaleb. Fast and secure global payments with Stellar. In *SOSP*, 2019.
- 35 G. Losa, E. Gafni, and D. Mazières. Stellar consensus by instantiation. In *DISC*, 2019.
- 36 E. MacBrough. Cobalt: BFT governance in open networks. *CoRR*, abs/1802.07240, 2018.
- 37 D. Malkhi, K. Nayak, and L. Ren. Flexible byzantine fault tolerance. In *CCS*, 2019.

- 38 D. Malkhi and M. Reiter. Byzantine quorum systems. In *STOC*, 1997.
- 39 D. Mazières. The Stellar consensus protocol: A federated model for internet-level consensus. <https://www.stellar.org>, April 2015.
- 40 A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song. The Honey Badger of BFT protocols. In *CCS*, pages 31–42, 2016.
- 41 S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.
- 42 B. Preneel. Collision resistance. In *Encyclopedia of Cryptography and Security*, 2011.
- 43 Protocol buffers. <https://developers.google.com/protocol-buffers/>, 2018.
- 44 D. Schwartz, N. Youngs, and A. Britto. The Ripple protocol consensus algorithm. Technical report, Ripple Labs Inc, 2014.
- 45 I. Sheff, R. van Renesse, and A. C. Myers. Distributed protocols and heterogeneous trust. *CoRR*, abs/1412.3136(arXiv:1412.3136), December 2014.
- 46 I. Sheff, X. Wang, H. Ni, R. van Renesse, and A. C. Myers. Charlotte: Composable authenticated distributed data structures, technical report, 2019.
- 47 I. Sheff, X. Wang, R. van Renesse, and A. C. Myers. Heterogeneous Paxos: Technical report, 2020.
- 48 H. Siu, Y. Chin, and W. Yang. Byzantine agreement in the presence of mixed faults on processors and links. *Parallel and Distributed Systems*, 9(4), April 1998.



# Multi-Threshold Asynchronous Reliable Broadcast and Consensus

Martin Hirt

Department of Computer Science, ETH Zürich, Switzerland  
hirt@inf.ethz.ch

Ard Kastrati

Department of Information Technology and Electrical Engineering, ETH Zürich, Switzerland  
akastrati@ethz.ch

Chen-Da Liu-Zhang

Department of Computer Science, ETH Zürich, Switzerland  
lichen@inf.ethz.ch

---

## Abstract

Classical protocols for reliable broadcast and consensus provide security guarantees as long as the number of corrupted parties  $f$  is bounded by a single given threshold  $t$ . If  $f > t$ , these protocols are completely deemed insecure. We consider the relaxed notion of *multi-threshold* reliable broadcast and consensus where validity, consistency and termination are guaranteed as long as  $f \leq t_v$ ,  $f \leq t_c$  and  $f \leq t_t$  respectively. For consensus, we consider both variants of  $(1 - \epsilon)$ -consensus and *almost-surely terminating* consensus, where termination is guaranteed with probability  $(1 - \epsilon)$  and 1, respectively. We give a very complete characterization for these primitives in the asynchronous setting and with no signatures:

- Multi-threshold reliable broadcast is possible if and only if  $\max\{t_c, t_v\} + 2t_t < n$ .
- Multi-threshold almost-surely consensus is possible if  $\max\{t_c, t_v\} + 2t_t < n$ ,  $2t_v + t_t < n$  and  $t_t < n/3$ . Assuming a global coin, it is possible if and only if  $\max\{t_c, t_v\} + 2t_t < n$  and  $2t_v + t_t < n$ .
- Multi-threshold  $(1 - \epsilon)$ -consensus is possible if and only if  $\max\{t_c, t_v\} + 2t_t < n$  and  $2t_v + t_t < n$ .

**2012 ACM Subject Classification** Theory of computation → Cryptographic protocols; Theory of computation → Distributed algorithms; Security and privacy → Cryptography

**Keywords and phrases** broadcast, byzantine agreement, multi-threshold

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2020.6

**Related Version** A full version of the paper is available at <https://eprint.iacr.org/2020/958>.

*This paper is eligible for best student paper award.*

## 1 Introduction

Consensus and reliable broadcast are fundamental building blocks in fault-tolerant distributed computing. Consensus allows a set of parties, each holding an input, to agree on a common value  $v'$ , where, if all honest parties hold the same input  $v$ ,  $v' = v$ . Reliable broadcast allows a designated party, called the sender, to consistently distribute a value  $v$  among a set of recipients such that all honest recipients output  $v$  in case the sender is honest. If the sender is dishonest, either all honest recipients output the same value or none of them terminates. Both primitives are used typically in the design of more complex applications, including multi-party computation, verifiable secret-sharing or voting, just to name a few.

The first consensus protocol was introduced in the seminal work of Lamport et al. [21] for the model where parties have access to a complete network of point-to-point authenticated channels, and where at most  $t < n/3$  parties are corrupted. Reliable broadcast was first



© Martin Hirt, Ard Kastrati, and Chen-Da Liu-Zhang;  
licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 6; pp. 6:1–6:16



Leibniz International Proceedings in Informatics

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

introduced by Bracha [6] as a useful primitive to construct building blocks in asynchronous environments. Since then, both primitives has been extensively studied in many different settings [7, 8, 6, 2, 25].

Most known fault-tolerant distributed protocols provide security guarantees in an *all-or-nothing* fashion: if up to  $t$  parties are corrupted, all security guarantees remain. However, if more than  $t$  parties are corrupted, the protocols do not provide any security guarantees. Multi-threshold protocols (also known as hybrid security) provide different security guarantees depending on the amount of corruption, thereby allowing a graceful degradation of security.

In this work, we consider consensus and reliable broadcast protocols with separate thresholds  $t_v$ ,  $t_c$  and  $t_t$  for *validity*, *consistency* and *termination*, respectively. For consensus, we consider both variants of  $(1 - \epsilon)$ -consensus and *almost-surely terminating* consensus, where termination is guaranteed with probability  $(1 - \epsilon)$  and 1, respectively.

Such multi-threshold primitives are not only of theoretical interest, but are also motivated by its use as core primitives in the design of more involved applications. In particular, they are used as a central building block in the recent line of works [26, 22], that leverage synchronous multi-party computation and consensus protocols to achieve *responsiveness*, where parties obtain output as fast as the network allows, given that the amount of corruption is low enough.

Our protocols work without the use of signatures and in the purely asynchronous model without the need to make any timing assumptions. Our contributions give a very complete picture of feasibility and impossibility results, which can be summarized as follows:

- Multi-threshold reliable broadcast is possible if and only if  $\max\{t_c, t_v\} + 2t_t < n$ .
- Multi-threshold almost-surely consensus is possible if  $\max\{t_c, t_v\} + 2t_t < n$ ,  $2t_v + t_t < n$  and  $t_t < n/3$ . The first two conditions are shown to be necessary as well. The question whether  $t_t < n/3$  is necessary is left as an open problem. However, we give a protocol assuming a global coin that does not require this condition.
- Multi-threshold  $(1 - \epsilon)$ -consensus is possible if and only if  $\max\{t_c, t_v\} + 2t_t < n$  and  $2t_v + t_t < n$ .

The impossibility proofs are simple and follow the lines of [9].

## 1.1 Related Work

There is a large literature devoted to achieving different types of hybrid security guarantees under different settings for agreement primitives and multi-party computation (MPC). We are only able to list an incomplete summary of related work.

The work in [11] provides constructions in the synchronous model for Byzantine broadcast with extended validity or consistency, where Byzantine broadcast is achieved up to a threshold  $t$ , and validity / consistency is achieved up to an extended threshold  $T \geq t$ , and then apply such constructions to achieve multi-party computation with full security up to  $t$  corruptions, and *unanimous abort* up to  $T \geq t$ . The above constructions exists if and only if  $t = 0$  or  $t + 2T < n$ . The works in [19, 20] focus on the question of achieving multi-party computation with full security under an honest majority, and *security with abort* under a dishonest majority. The line of works in [13, 24] provide constructions that achieve trade-offs that include information-theoretic security up to a certain threshold, and computational security up to a larger threshold, with different types of guarantees. A different line of works provide security against different types of corruption (also known as mixed adversaries). The works [12, 18] consider multi-party computation protocols where security holds even when up to  $t_p$ ,  $t_f$ ,  $t_a$  parties can be passively, fail-stop, actively corrupted, respectively. Finally, there are works that combine mixed adversaries with hybrid security [16, 17, 15].

A recent line of works [23, 26, 22] achieve trade-offs between *responsiveness*, where parties obtain output as fast as the network allows, and other security guarantees, for consensus, SMR and MPC, assuming a synchronous network. The work [14] considers to networks that tolerate some level of disconnection between the parties, as long as there is a connected component with an honest majority of the parties. Finally, the works [3, 4, 5] provide protocols that achieve security guarantees under a synchronous network up to  $t_s$  corruptions, and under an asynchronous network up to  $t_a$  corruptions.

## 1.2 Comparison to Prior Work

As mentioned above, some works use as building blocks multi-threshold asynchronous consensus and reliable broadcast primitives. In particular, the works in [23, 14, 22] make use of an asynchronous multi-threshold consensus protocol with increased validity and consistency. Their constructions differ from ours in two aspects: 1) they operate in a setting where parties have access to a public-key infrastructure and 2) their constructions inherently require that the termination threshold is below  $n/3$ .

The constructions for consensus and reliable broadcast in [3, 4] considers different thresholds. In [3], the authors design a consensus protocol with increased *validity with termination* (where validity also ensures termination in case of pre-agreement) assuming a global common coin, based on the protocol in [25]. Similarly, in [4], the authors provide a construction for reliable broadcast with two thresholds allowing for validity with termination in the honest sender case, and consistency with *reliable termination* (where either all honest parties terminate or none), in the dishonest sender case. We provide constructions without assuming a global coin, which in addition allow to have the termination threshold above validity and consistency.

## 2 Model

We consider a setting in which parties have access to a complete network of authenticated channels. The adversary has full control over the network and can schedule the messages in an arbitrary manner. However, each message must be eventually delivered. Moreover, we consider the setting where parties do not have any setup available.

We consider an *adaptive* adversary who can gradually corrupt parties and take full control over them. Note, however, that our impossibility proofs hold even against a *static* adversary that is assumed to choose the corrupted parties at the beginning of the protocol execution. We require our protocols to be *unconditionally secure*, meaning that security holds even against a computationally unbounded adversary. On the other hand, our impossibility results hold even against a computationally bounded adversary.

In the protocols we say that a party terminates when it stops participating in the protocol. Note that we distinguish between outputting and terminating, in the sense that a party might output a value but still continue participating.

## 3 Multi-Threshold Reliable Broadcast

Reliable broadcast is a fundamental primitive in distributed computing which allows a sender to consistently distribute a message towards a set of recipients. We consider a setting with  $n + 1$  parties, one sender  $S$  and  $n$  recipients  $\mathcal{R} = \{R_1, \dots, R_n\}$ . Let us denote the number of corrupted recipients (not including the sender) at the end of the protocol execution by  $f$ .

► **Definition 1 (Reliable Broadcast).** Let  $\mathcal{M}$  be a finite message space and  $f$  be the number of corrupted recipients at the end of the execution. A protocol  $\pi$  where initially the sender  $S$  has an input  $m \in \mathcal{M}$  and every recipient  $R_i$  upon termination outputs  $m_i \in \mathcal{M}$ , is a reliable broadcast protocol, with respect to thresholds  $t_c$ ,  $t_v$ , and  $t_t$ , if it satisfies the following:

- **Consistency.** If  $f \leq t_c$ , then every honest recipient that terminates outputs the same message. That is,  $\exists m' \in \mathcal{M} : \forall$  honest  $R_i$  that terminate  $m_i = m'$ .
- **Validity.** If  $f \leq t_v$  and the sender is honest, then every honest recipient  $R_i$  that terminates outputs the sender's message. That is,  $\forall$  honest  $R_i$  that terminate  $m_i = m$ .
- **Termination.**
  1. An honest sender always terminates.
  2. If  $f \leq t_t$  and an honest recipient terminates, then every honest recipient eventually terminates.
  3. If  $f \leq t_t$  and the sender is honest, then eventually every honest recipient terminates.

### 3.1 Protocol

We present a reliable broadcast protocol with respect to thresholds  $t_c$ ,  $t_v$  and  $t_t$ , as long as  $\max\{t_v, t_c\} + 2t_t < n$ . The protocol is a generalization of Bracha's broadcast protocol [6].

#### Protocol $\Pi_{\text{rbc}}^{t_c, t_v, t_t}$

The sender  $S$  holds input  $m \in \mathcal{M}$ . Upon termination every recipient  $R_i \in \mathcal{R}$  outputs a message.

- **Code for the sender  $S$** 
  1. Send the message (MSG,  $m$ ) to all recipients in  $\mathcal{R}$  and terminate.
- **Code for recipient  $R_i \in \mathcal{R}$** 
  1. Upon receiving *first* (MSG,  $m$ ) from the sender, send (ECHO,  $m$ ) to all recipients.
  2. Upon receiving (ECHO,  $m'$ ) from  $n - t_t$  parties that agree on the value  $m' \in \mathcal{M}$ , send (READY,  $m'$ ) to all recipients.
  3. Upon receiving (READY,  $m'$ ) from  $\max\{t_v, t_c\} + 1$  parties that agree on the value  $m' \in \mathcal{M}$ , send (READY,  $m'$ ) to all recipients.
  4. Upon receiving (READY,  $m'$ ) or (TERMINATE) from  $n - t_t$  parties from which at least  $\max\{t_v, t_c\} + 1$  are READY messages and (they all) agree on the value  $m' \in \mathcal{M}$ , send (TERMINATE), output  $m'$  and terminate.

► **Lemma 2.** If  $f \leq \max\{t_v, t_c\}$ ,  $\forall m' \in \mathcal{M}$  the first honest recipient that sends (READY,  $m'$ ) received at least  $n - t_t$  (ECHO,  $m'$ ) messages.

**Proof.** For any  $m' \in \mathcal{M}$ , (READY,  $m'$ ) messages are sent by honest recipients either in line 2 or 3 of the recipient's code. However,  $\forall m' \in \mathcal{M}$  the first (READY,  $m'$ ) message can only be sent in line 2. This is due to the fact that if  $f \leq \max\{t_v, t_c\}$ , line 3 implies that there must be some other honest recipient that previously sent a (READY,  $m'$ ) message too. ◀

► **Lemma 3.** If  $f \leq \max\{t_v, t_c\}$ , the messages (READY,  $m'$ ) sent by honest recipients are consistent. That is, there  $\exists m'' \in \mathcal{M}$  such that for every honest recipient  $R_i$  that sends (READY,  $m'$ ),  $m' = m''$ .

**Proof.** Suppose not; let  $R_i$  and  $R_j$  be the *first* honest recipients that send (READY,  $m'$ ) and (READY,  $m''$ ) with  $m' \neq m''$ . Due to Lemma 2,  $R_i$  received at least  $n - t_t$  (ECHO,  $m'$ ) messages whereas  $R_j$  received at least  $n - t_t$  (ECHO,  $m''$ ) messages. It follows, at least  $2(n - t_t) - n = n - 2t_t > \max\{t_v, t_c\}$  players dishonestly sent inconsistent ECHO messages to  $R_i$  and  $R_j$ . However, each honest recipient sends an ECHO message at most once and there are at most  $f \leq \max\{t_v, t_c\}$  dishonest recipients. A contradiction. ◀



► **Lemma 4.** *If  $f \leq \max\{t_v, t_c\}$  and an honest sender broadcasts  $m$ , for every honest recipient that sends (READY,  $m'$ ),  $m' = m$ .*

**Proof.** Suppose not; Let  $R_i$  be the first honest recipient that sends (READY,  $m'$ ) with  $m' \neq m$ . Due to Lemma 2,  $R_i$  received at least  $n - t_t$  (ECHO,  $m'$ ) messages. However, in case the sender is honest and broadcasts  $m$  every honest recipient will ECHO only the sender's value ( $m$ ). Hence there can be at most  $f \leq \max\{t_v, t_c\} < n - t_t$  (ECHO,  $m'$ ) with  $m \neq m'$ . A contradiction. ◀

► **Lemma 5.** *If  $f \leq t_t$  and an honest recipient terminates, then every honest recipient eventually terminates.*

**Proof.** Let  $R_i$  be the first honest recipient that terminates. Then,  $R_i$  received (READY,  $m'$ ) or (TERMINATE) from  $n - t_t$  messages from which at least  $\max\{t_v, t_c\} + 1$  are READY messages. Furthermore, all READY messages agree on  $m'$ . Under the assumption that no other honest recipient has terminated so far, we know that no (TERMINATE) messages were sent from the honest recipients. Hence, by taking into account that  $n - t_t > \max\{t_v, t_c\} + t_t$  and  $f \leq t_t$ , it follows that at least  $\max\{t_v, t_c\} + 1$  recipients have sent (READY,  $m'$ ) to all other parties. Every honest recipient will eventually either receive these (READY,  $m'$ ) messages and send a (READY,  $m'$ ) as well or terminate before receiving them and send a (TERMINATE) message instead. Since there are at least  $n - t_t$  honest recipients, it follows that eventually every honest recipient  $R_l$  that didn't terminate yet will receive  $n - t_t$  messages (READY,  $m'$ ) or (TERMINATE) from which at least  $\max\{t_v, t_c\} + 1$  are READY messages and they all agree on  $m'$ . Thus, every honest  $R_l$  eventually terminates as well. ◀

► **Lemma 6.** *If  $f \leq t_t$  and the sender is honest, at least one honest recipient eventually terminates.*

**Proof.** Since every honest recipient echoes the sender's value, there will be at least  $n - t_t$  (ECHO,  $m$ ) messages. Similarly, since there are  $n - t_t$  (ECHO,  $m$ ) messages in the network, every honest recipient will eventually send a (READY,  $m$ ). Finally, since there are  $n - t_t$  (READY,  $m$ ) messages in the network, at least one honest recipient will terminate. ◀

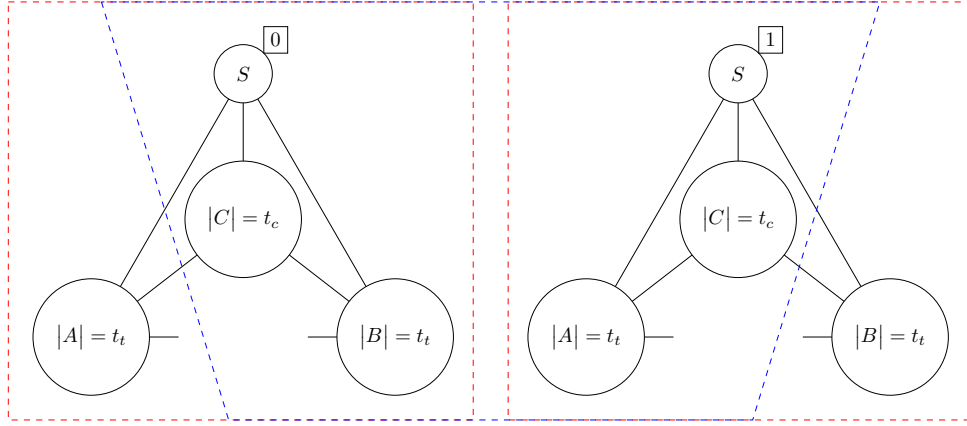
► **Theorem 7.** *Let  $0 \leq t_c, t_v, t_t < n$ .  $\Pi_{\text{rbc}}^{t_c, t_v, t_t}$  is a multi-threshold broadcast according to Definition 1 if  $\max\{t_v, t_c\} + 2t_t < n$ .*

**Proof.**

- *Consistency & Validity.* Assume  $f \leq \max\{t_v, t_c\}$ . Every honest recipient that outputs a message, has received at least  $\max\{t_v, t_c\} + 1$  (READY,  $m$ ). Since  $f \leq \max\{t_v, t_c\}$ , it follows at least one is sent from an honest party. From Lemma 3 we know that READY messages are consistent, hence we achieve consistency. From Lemma 4 we know that the READY messages from honest parties contain only the sender's value, hence we achieve validity.
- *Termination.* We prove the three termination properties from the Definition 1.
  1. For the first requirement, it is trivial to see that an honest sender always terminates.
  2. The second requirement is proven in Lemma 5.
  3. For the third requirement, from Lemma 6 we know that if the sender is honest, at least one honest recipient terminates. We can apply Lemma 5 again, and see that every honest recipient terminates. ◀

### 3.2 Impossibility Proofs

In this section, we show that the protocol  $\Pi_{\text{rbc}}^{t_c, t_v, t_t}$  presented above is optimal. That is, there is no reliable broadcast protocol with  $\max\{t_v, t_c\} + 2t_t \geq n$ , where  $t_c, t_v, t_t \geq 0$ . We prove each bound separately.



■ **Figure 1** The same configuration can be viewed as: a) (in red) two independent runs of the protocol on the left and the right side, where messages between  $A$  and  $B$  are delayed: One where the sender has input 0 and one where the sender has input 1; b) (in blue) the sender and the set  $C$  is corrupted and behaves differently towards  $A$  and  $B$ .

► **Theorem 8.** *There exists no multi-threshold broadcast protocol with  $t_c + 2t_t \geq n$ .*

**Proof.** Suppose not; let  $\pi$  be a multi-threshold broadcast protocol with  $t_c + 2t_t = n$ . We partition the set of all recipients into three sets  $A$ ,  $B$  and  $C$  with size  $|A| = |B| = t_t$  and  $|C| = t_c$ . We build the network as in Figure 1.

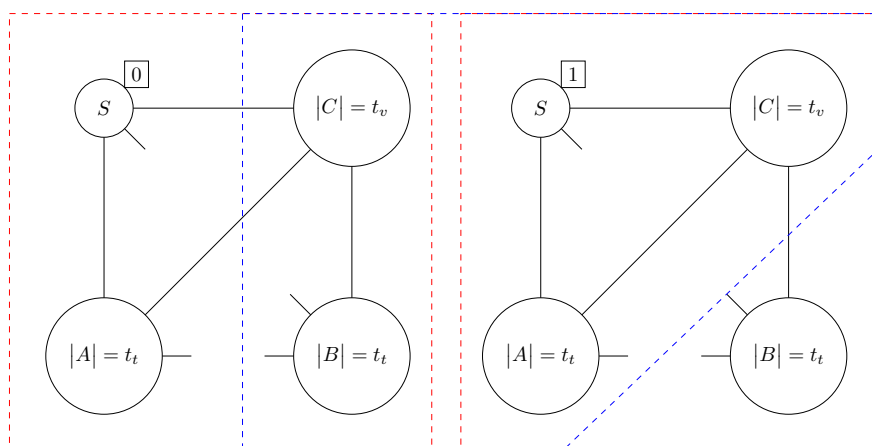
Figure 1(in red) on the left side, we can see an independent run where all parties are honest and messages between  $A$  and  $B$  are delayed by the scheduler. Since  $A$  and  $B$  are of size  $t_t$ , all parties terminate with an output. Moreover, since all parties are honest, the output is 0. In particular, parties in  $A$  output 0. Similarly,  $B$  outputs 1 on the right side.

Now consider an attacker that corrupts the sender and  $C$ , and emulates the protocol as in the scenario in Figure 1(in blue). Since this configuration is exactly the same as the red one,  $A$  outputs 0 and  $B$  outputs 1. This results in a contradiction to the consistency property of the multi-threshold broadcast. ◀

► **Theorem 9.** *For any  $t_v > 0$ , there is no multi-threshold broadcast protocol with  $t_v + 2t_t \geq n$ .*

**Proof.** Suppose not; let  $\pi$  be a multi-threshold broadcast protocol with  $t_v + 2t_t = n$ . We partition the set of all recipients into three sets  $A$ ,  $B$  and  $C$  with  $|A| = |B| = t_t$  and  $|C| = t_v$ . We build the a configuration as in Figure 2.

Consider Figure 2(in red), on the left side, where messages between  $S$  and  $B$ , or between  $A$  and  $B$  are delayed. Since  $B$  is of size  $t_t$ , all parties must output a value *without* waiting for the messages from  $B$  (as  $B$  could be corrupted). Moreover, since all parties are honest,  $A$  and  $C$  output 0. Furthermore, since  $C$  outputs 0, because of the second requirement of the termination of broadcast – *if one recipient terminates, then every recipient terminates* –  $B$  outputs 0 as well. Note that  $A$  and  $S$  have together size  $t_t + 1$ , but the second requirement of termination requires  $B$  to terminate even if the sender is corrupted. The same argument can be applied on the right side of Figure 2(in red).



■ **Figure 2** The same configuration can be viewed as: a) (in red) two independent runs of the protocol on the left and the right side, where messages between  $A$  and  $B$  are delayed, and also between  $S$  and  $B$ . One where the sender has input 0 and one where the sender has input 1; b) (in blue) the set  $C$  is corrupted and behaves differently towards  $S$  and  $A$  on the left side and differently towards  $B$  on the right side.

Now, consider an attacker that corrupts the parties in  $C$  and emulates the protocol as in the scenario in Figure 2 (in blue). Because both scenarios are the same setup,  $A$  outputs 0 on the left side, whereas  $B$  outputs 1 on the right side. Thus, validity is violated. ◀

#### 4 Almost-Surely Multi-Threshold Consensus

Stated in simple terms, consensus allows a set of parties to agree on a common value. More formally, the protocol starts with every party having an input and ends with every party having a consistent output. Moreover, if every honest party starts with the same input, they keep it. Due to the FLP impossibility proof [10], non-terminating executions are inevitable for every consensus protocol. Hence, we require the parties to terminate only with probability 1, termed in the literature as *almost-surely terminating* consensus.

► **Definition 10 (Consensus).** Let  $\mathcal{M}$  be a finite message space and  $f$  be the number of corrupted parties at the end of the execution. A protocol  $\pi$  where initially each party has an input  $x_i \in \mathcal{M}$  and finally every party  $P_i$  upon termination has an output  $y_i \in \mathcal{M}$ , is a consensus protocol, with respect to thresholds  $t_c, t_v, t_t$ , if it satisfies the following:

- **Consistency.** If  $f \leq t_c$ , then the output of every honest party is the same value. That is,  $\exists y \in \mathcal{M} : \forall$  honest  $P_i$  that output  $y_i = y$ .
- **Validity.** If  $f \leq t_v$  and every honest party has the same input value  $x \in \mathcal{M}$ , then the output of every honest party  $P_i$  is  $x$ . That is,  $\forall$  honest  $P_i$  that output  $y_i = x$ .
- **Termination.** If  $f \leq t_t$ , then with probability 1 eventually every honest party outputs and terminates.

In the following, we present a multi-threshold consensus protocol with respect to thresholds  $t_c, t_v$  and  $t_t$ , where  $\max\{t_c, t_v\} + 2t_t < n$ ,  $2t_v + t_t < n$  and  $t_t < n/3$ . In the full version, we also show that the bounds  $\max\{t_c, t_v\} + 2t_t < n$  and  $2t_v + t_t < n$  are required. We leave the feasibility of almost-surely multi-threshold consensus with  $t_t \geq n/3$  as an open question. However, in the full version we provide a construction that overcomes the  $n/3$  bound for the case where parties have access to a global coin.

The protocol is an adaptation of Bracha's consensus [6] protocol. The main idea of Bracha's consensus is to use a reliable broadcast primitive and build a *correctness enforcement* scheme, where only messages that are intended by the protocol design are accepted. The only difference in our protocol is that we plug our multi-threshold broadcast protocol described in the previous section into the correctness enforcement scheme proposed by Bracha. We choose to use a multi-threshold reliable broadcast  $\Pi_{\text{rbc}}^{t_s, t_s, t_t}$  that achieves validity and consistency up to  $t_s = n - 2t_t - 1$  corruptions (which is achievable since  $t_s + 2t_t < n$ ). Note that  $t_t < n/3$  implies that  $t_s \geq t_t$ .

#### 4.1 Multi-Threshold Correctness Enforcement

For completeness and readability of our protocols, we include a summary of the correctness enforcement mechanism. Further details can be found in [6]. The following constructions and proof techniques are very similar to [6] with the only difference that we plug our multi-threshold broadcast protocol  $\Pi_{\text{rbc}}^{t_s, t_s, t_t}$ . Furthermore, we assume  $t_s \geq t_t$ .

**Round-Based Asynchronous Protocols.** We consider protocols that are composed by *rounds*. In each round  $k$ , every party uses the multi-threshold broadcast protocol to send a value to all parties. Subsequently, every party waits to receive a set  $S$  of the values (of size at most  $n - t_t$ ) and computes a new value according to some function  $F^k(\cdot)$  for the next round.

**Validation Sets.** Each party  $P_i$  keeps for each round  $k$  a set of values  $\mathcal{V}_i^k$ , called a *validation set*, with the values that are broadcast in round  $k$ . Each value  $x_i$  broadcasted in round  $k$  by  $P_i$  is stored as  $(P_i, k, x_i)$ . When a value is broadcast by some party at round  $k + 1$ , every party checks locally whether there exists a subset of values in  $\mathcal{V}_i^k$  that explains the broadcast value. That is,  $\mathcal{V}_i^k$  is defined as follows:

- For  $k = 1$ ,  $(P_j, 1, x_j) \in \mathcal{V}_i^1$  if  $x_j$  is received by  $P_i$  from a multi-threshold broadcast protocol with sender  $P_j$  at round 1.
- For  $k > 1$ ,  $(P_j, k, x_j) \in \mathcal{V}_i^k$ , if  $x_j$  is received by  $P_i$  from a multi-threshold broadcast protocol with sender  $P_j$  at round  $k$ , and there is a subset  $S \subseteq \mathcal{V}_i^{k-1}$  such that  $x_j = F^{k-1}(S)$ .

We say a party  $P_i$  *validates* a message  $x_j$  in round  $k$  if  $(P_j, k, x_j) \in \mathcal{V}_i^k$ . The parties update their  $\mathcal{V}$  sets whenever they validate a message. If a party  $P_i$  outputs a value during a broadcast protocol but the message is still not validated it is ignored in the protocol, although it is stored for future validation.

**Protocol** A round with correctness enforcement

**Code for the party  $P_i$  with input  $x_i$  at round  $k$**

1. Multi-Threshold Broadcast( $x_i$ ) to all the parties.
2. Wait until a set  $S$  of messages have been validated.
3. Set  $x_i = F^k(S)$ .

We state a list of lemmas that are guaranteed from the correctness enforcement mechanism. The proofs will appear in the full version of the paper.

► **Lemma 11.** *If  $f \leq t_s$ , in every round  $k$  of the protocol the added values in the validation sets of all honest parties are consistent. That is:*

$$\forall \text{ honest } P_i, P_j : \forall P_l : (P_l, k, x_l) \in \mathcal{V}_i^k \wedge (P_l, k, \tilde{x}_l) \in \mathcal{V}_j^k \implies x_l = \tilde{x}_l$$

► **Lemma 12.** *If  $f \leq t_s$  and an honest sender  $P_i$  broadcasts  $(P_i, k, x_i)$  in a round  $k$  of the protocol, the validation sets of all honest parties contain only the sender's value. That is:*

$$\forall \text{ honest } P_i, P_j : (P_i, k, x_i) \in \mathcal{V}_j^k \implies P_i \text{ broadcast } x_i \text{ in round } k.$$

► **Lemma 13.** *If  $f \leq t_t$  every party will eventually go from round  $k$  to round  $k + 1$ .*

## 4.2 Protocol

We describe the protocol, which is a generalization of Bracha's consensus [6]. The protocol executes in parallel the two sub-protocols 'Reaching agreement' and 'Termination'.

### Protocol $\Pi_{\text{as-con}}^{t_c, t_v, t_t}$

*Input.* Every party  $P_i$  holds input  $x_i$ .

*Variable.*  $y_i = \perp$ .

#### Reaching agreement.

##### ■ Code for the party $P_i$ at phase $k$ .

1.  $\Pi_{\text{rbc}}^{t_s, t_s, t_t}(x_i)$ . Wait until  $n - t_t$  messages are validated.
  - $x_i =$  majority of the validated elements.
2.  $\Pi_{\text{rbc}}^{t_s, t_s, t_t}(x_i)$ . Wait until  $n - t_t$  messages are validated.
  - If all of the validated messages have the same value  $x$ ,  $x_i = (\text{propose}, x)$
  - Otherwise, keep the same  $x_i$ .
3.  $\Pi_{\text{rbc}}^{t_s, t_s, t_t}(x_i)$ . Wait until  $n - t_t$  messages are validated.
  - a. If at least  $n - t_t$  of the validated messages have the same value  $(\text{propose}, x)$ , then update  $y_i = x$  and run the 'Reaching agreement code' for only one more phase.
  - b. Else if at least  $t_t + 1$  of the validated messages have the same value  $(\text{propose}, x)$ , then  $x_i = x$ .
  - c. Otherwise, choose 0 or 1 with probability 1/2 for  $x_i$  (coin toss).
4. Go to phase  $k + 1$ .

#### Termination.

- Upon updating  $y_i$ , send  $(\text{READY}, y_i)$  to all parties.
- Upon receiving  $(\text{READY}, m')$  messages from  $\max\{t_c, t_v\} + 1$  parties that agree on the value  $m'$ , send  $(\text{READY}, m')$  to all parties.
- Upon receiving  $(\text{READY}, m')$  from  $n - t_t$  parties that agree on the value  $m'$ , output  $m'$  and terminate.

► **Lemma 14.** *If  $f \leq t_s$ , it is impossible for an honest party to propose 0 and an honest party to propose 1 in the same phase  $k$ .*

**Proof.** The proof is by contradiction. Suppose parties  $P_i$  and  $P_j$  propose 0 and 1, respectively, in phase  $k$ . Thus in line 2 of phase  $k$ ,  $P_i$  validated  $n - t_t$  messages with value 0 and  $P_j$  validated  $n - t_t$  messages with value 1. Since  $n - 2t_t > 0$ , it follows that  $P_i$  and  $P_j$  have inconsistent messages in their validation sets, which contradicts Lemma 11. ◀

We say that a phase  $k$  is  $x$ -fixed (for  $x \in \{0, 1\}$ ), if honest parties that starts phase  $k$  validate only  $x$  as an input value broadcast by any party.

► **Lemma 15.** *If  $f \leq t_s$  and an honest party  $P_i$  updates  $y_i = x \in \{0, 1\}$  at some phase  $k$ , phase  $k + 1$  is  $x$ -fixed.*

**Proof.** Suppose that some party  $P_i$  updates  $y_i = x \in \{0, 1\}$  at phase  $k$ .  $P_i$  must have validated at least  $n - t_t$  proposals for  $x$  in step 3 of phase  $k$ . Let  $P_j$  be any party that starts phase  $k + 1$ . In phase  $k$ , since  $P_i$  validated  $n - t_t$  proposals for  $x$  and  $t_t < n/3$ ,  $P_j$  must have

validated at least  $t_t + 1$  for  $x$ . Moreover by Lemma 14,  $P_j$  does not validate any proposals for  $x' \neq x$ . So any  $P_j$  can only set its variable  $x_j$  to  $x$  in step 3. Hence, no honest party can validate  $x' \neq x$  in the next phase as an input. Therefore, phase  $k + 1$  is  $x$ -fixed. ◀

► **Lemma 16.** *If a phase  $k$  is  $x$ -fixed then every honest party  $P_i$  that reaches step 3 of the phase  $k$  updates  $y_i = x$  at the end of the phase.*

**Proof.** Suppose a phase  $k$  is  $x$ -fixed. Then, all honest parties validate only  $x$  as an input value. Hence, every honest party can only propose  $x$  as their input. Consider a party  $P_i$  that reaches step 3 of phase  $k$ . Clearly,  $P_i$  can only validate proposals with  $x$ . Hence, from line 3 of the *reaching agreement* part of the protocol we can see that  $P_i$  updates  $y_i = x$ . ◀

► **Lemma 17 (Liveness).** *If  $f \leq t_t$  and if no honest party updated  $y_i$ , the parties will eventually go from phase  $k$  to phase  $k + 1$ .*

**Proof.** Immediate from Lemma 13. ◀

► **Lemma 18.** *If  $f \leq t_t$  every honest party with probability 1 will update  $y_i$  to the same value.*

**Proof.** First note that we assume  $t_t \leq t_s$ . By Lemma 17, as long as no party updates  $y_i$ , honest parties don't get stuck in any round. Every honest party that doesn't update  $y_i$  in phase  $k$ , sets its value  $x_i$  for the next phase either based on step 3(ii) or step 3(iii). Let  $P_i$  be the first honest party that completed round  $3k + 3$ . There are two cases:

- Party  $P_i$  has validated at least one (propose,  $x$ ). With probability  $p \geq 1/2^{n-t_t}$  all honest parties that toss a coin choose  $x$ . By Lemma 14, the remaining honest parties that set their value deterministically, are forced to set their value to  $x$ .
- Party  $P_i$  has validated no (propose,  $x$ ). Since  $P_i$  validated  $n - t_t$  messages and  $t_t < n/3$ , no other honest party  $P_j$  can validate more than  $t_t$  values of the form (propose,  $x$ ). Hence, every honest party tosses a coin. The probability that every honest party tosses the same value is again  $p \geq 1/2^{n-t_t}$ .

Hence, in either case after each phase the probability that honest parties have the same value is greater or equal then  $1/2^{n-t_t}$ . If at some phase  $k$  every honest party has the same value then it follows that in the next round there can be at most  $t_t$  parties will input  $\bar{x}$  (the ones that maliciously change the outcome of the local coin). However, by Lemma 15 (note that we have  $t_t \leq t_s$ ) and since  $n - 2t_t > t_t$ , it follows that the majority of each subset of size  $n - t_t$  in the next round will result in  $x$ . Hence, next phase is  $x$ -fixed. By Lemma 16 it follows that every honest party will update  $y_i$  after that phase. Hence, after round  $k$  the probability of not updating  $y_i$  is  $(1 - p)^k$ . As  $k$  goes to infinity, the probability goes to 0. ◀

► **Theorem 19.** *Let  $0 \leq t_c, t_v, t_t < n$ .  $\Pi_{\text{as-con}}^{t_c, t_v, t_t}$  is an almost-surely terminating multi-threshold consensus (see Definition 10) if  $\max\{t_c, t_v\} + 2t_t < n$ ,  $2t_v + t_t < n$  and  $t_t < n/3$ .*

**Proof.** ■ *Validity.* Suppose all honest parties have the same input  $x$ . In the first round, since there are at most  $f \leq t_v$ , at most  $t_v$  elements with value  $\bar{x}$  can be broadcast by corrupted parties. By Lemma 15 (note that  $t_v \leq \max\{t_c, t_v\} = t_s$ ) and since  $n - t_t - t_v > t_v$ , it follows that honest parties can only validate  $x$  as the outcome of the first round. Hence, the first phase is  $x$ -fixed. Due to Lemma 16, it follows that every honest party updates  $y_i = x$  at the end of the first phase. Furthermore, by applying Lemma 15 and 16 recursively, one can easily see that once a phase is  $x$ -fixed it always remains so. Hence, parties can never change the value. Furthermore, in the termination part of the protocol it is not hard to see that READY messages are unique (see Section 3 for a detailed proof) and contain only  $x$ . Hence, every honest party that outputs, outputs  $x$ .

- *Consistency.* Suppose some party  $P_i$  and  $P_j$  update different values  $y_i = x$  and  $y_j = x'$  in phase  $k$  and  $k'$  respectively. There are two cases:
  1.  $k = k'$ . Since a party can update a value in phase  $k$  only if that value was proposed in round  $k$ , it follows both  $x$  and  $x'$  were proposed in phase  $k$ . By Lemma 14 this is impossible.
  2.  $k < k'$ . Since  $P_i$  updates  $y_i$  in phase  $k$  then due to Lemma 15 phase  $k + 1$  is  $x$ -fixed. Similar to previous arguments, by applying Lemma 15 and 16 recursively, one can easily see that once a phase is  $x$ -fixed it always remains so. It follows that all parties can only update  $y_i = x$  in the next phases.

This proves that the update of  $y_i$  by all parties is unique. In return this implies that READY messages are unique as well. By similar arguments as in Section 3, it is not hard to see that the consistency is achieved also in the termination part of the protocol.

- *Almost-Surely Termination.* From Lemma 18 it follows that with probability 1 every honest party will update  $y_i$  to the same value (say  $x$ ). Since, after updating  $y_i = x$  parties take part only in one more round, with probability 1 every honest party will “get out” of the infinite loop. By simply setting the message space  $\mathcal{M} = \{0, 1\}$  one can easily prove now termination similar to Section 3. It follows, like in multi-threshold broadcast protocol, every party will eventually send the same (READY,  $x$ ) with  $x \in \{0, 1\}$ . Hence eventually there will be  $n - t_t$  (READY,  $x$ ) messages that agree on a value and thus at least an honest party with probability 1 terminates. Again, similar to broadcast one can see that if an honest party terminates, then every honest party eventually terminates. Thus with probability 1 eventually everyone terminates. ◀

## 5 (1 - $\epsilon$ ) Multi-Threshold Consensus

The general idea in the previous section is to use randomness such that by chance the parties reach agreement. Once they do, agreement is preserved. However, in the regime where  $t_t \geq n/3$ , the following challenges arise. First, note that if  $t_t \geq n/3$  and  $\max\{t_c, t_v\} + 2t_t < n$ , then  $\max\{t_c, t_v\} < t_t$ . As a consequence, there is a region where the multi-threshold broadcast protocol only guarantees termination, but does not guarantee the consistency and validity of the outputs. This is problematic, because the adversary can change the outputs of each broadcast instance such that no messages are validated in the correctness enforcement scheme. As a consequence, parties get stuck in a phase and never terminate. The second challenge is with respect to the coin. If  $t_t \geq n/3$ , even when all honest parties obtain the same value  $v$  as local coin, the adversary can schedule messages so that the majority decision among  $n - t_t$  values is inconsistent among the parties. Finally, as pointed out in [1], the correctness proof for  $n/3 \leq t_t < n/2$  is more subtle and requires reasoning about two consecutive phases. Moreover, they show that the global-variant of Ben-Or *doesn't work* for the case  $n/3 \leq t_t < n/2$ .

We overcome the first challenge by plugging in a *detectable broadcast* primitive into the correctness enforcement mechanism, which allows parties to eventually detect misbehavior in the case where they obtain different values. The second challenge is resolved by cycling through all sets  $S$  of  $t_t + 1$  parties, where only parties in  $S$  sample a random coin. This way, if all parties in  $S$  are honest and chooses the same local coin, then everyone adopts the same value. Finally, the last challenge is resolved by adding one round for each phase, which allows to analyse the phases independently of each other.



With these techniques, we construct a  $(1 - \epsilon)$  multi-threshold consensus protocol if  $\max\{t_c, t_v\} + 2t_t < n$  and  $2t_v + t_t < n$ , where termination holds with probability  $(1 - \epsilon)$  (instead of 1). The bounds are optimal (see the full version). In contrast to the almost-surely-terminating version, it is possible to achieve this notion with  $t_t \geq n/3$ . The number of rounds depends on the error  $\epsilon$ . For negligible  $\epsilon$ , the number of rounds is exponential in  $n$ .

► **Definition 20** ( $(1 - \epsilon)$ -Consensus). *The consistency and validity property of  $(1 - \epsilon)$ -consensus are the same as in Definition 10. We only change the termination property.*

■ **Termination.** *If  $f \leq t_t$ , then with probability  $1 - \epsilon$  eventually every honest party outputs and terminates.*

## 5.1 Detectable Correctness Enforcement

As mentioned, if  $t_t > \max\{t_v, t_c\}$ , in the region where the adversary corrupts  $\max\{t_v, t_c\} < f \leq t_t$ , the multi-threshold broadcast does not guarantee any consistency among messages, allowing the adversary to make parties reach a state where no messages are validated and thus all honest parties get stuck. Instead, we use a *detectable* multi-threshold broadcast, which guarantees consistent outputs when  $f \leq \max\{t_c, t_v\}$  as in multi-threshold broadcast, but in addition allows parties to detect potential misbehavior if  $f \leq t_t$ . Note, however, that we don't require termination, i.e., parties may need to run forever. The protocol is based on the one in Section 3, so we defer its description and analysis to the full version. Plugging the detectable multi-threshold broadcast in the correctness enforcement results in detectable correctness enforcement, where the properties of correctness enforcement hold or parties detect Byzantine behavior.

► **Definition 21 (Detectable Multi-Threshold Broadcast).** *Let  $\mathcal{M}$  be a finite message space and  $f$  be the number of corrupted parties. A protocol  $\pi$ , where initially the sender  $S$  has an input message  $m \in \mathcal{M}$  and subsequently every recipient  $R_i \in \mathcal{R}$  potentially outputs a message  $m_i \in \mathcal{M}$  and/or a detection flag DETECT, is a detectable multi-threshold broadcast protocol with respect to thresholds  $t_c$ ,  $t_v$  and  $t_t$ , if it satisfies the following:*

- **Consistency.** *If  $f \leq t_c$ ,  $\exists m' \in \mathcal{M} : \forall$  honest  $R_i$  that output the message  $m_i$ , the value of  $m_i = m'$ . Furthermore, no honest recipient  $R_i$  outputs the detection flag DETECT.*
- **Validity.** *If  $f \leq t_v$  and the sender is honest,  $\forall$  honest  $R_i$  that output the message  $m_i$ , the value of  $m_i = m$ . Furthermore, no honest recipient  $R_i$  outputs the detection flag DETECT.*
- **Totality-or-Detection.**
  1. *If  $f \leq t_t$  and an honest recipient outputs the message  $m' \in \mathcal{M}$  then eventually every honest recipient outputs the message  $m'$  or every honest recipient outputs the detection flag DETECT.*
  2. *If  $f \leq t_t$  and the sender is honest, then eventually every honest recipient outputs the message  $m$  or every honest recipient outputs the detection flag DETECT.*

**Notation.** We say that “ $P_i$  detects Byzantine behaviour” to denote that  $P_i$  outputs the detection flag DETECT in an execution of detectable multi-threshold broadcast. Note that detectable multi-threshold broadcast guarantees that either all honest recipients eventually output DETECT, or none of them does.

## 5.2 Common Coin

In the protocol from Section 4, parties toss a coin until they reach by chance agreement. If  $t_t \geq n/3$ , the adversary can maliciously change the local coins for some of the parties and break termination. We show a protocol that allows parties to output the same value, even if



the adversary changes the outcome of the local coin of some of the parties. If  $t_t < n/3$ , this is easily done by allowing each party to broadcast a random value and choosing the majority among the  $n - t_t$  values that are received. For  $n/3 \leq t_t < n/2$ , one cannot take the majority value. The idea is to let only a subset  $R$  of  $t_t + 1$  parties toss a local coin:

**Protocol**  $\text{Coin}(R)$ ,  $f \leq t_t < n/2$ ,  $|R| = t_t + 1$

1. If  $P_i \in R$ : choose 0 or 1 with probability 1/2 and detectable-broadcast the outcome to everyone.
2. Every party outputs the value of the first broadcast that output.

If all the parties in  $R$  are honest and toss the same coin, then every party outputs the same value. During the consensus protocol, we *cycle* through all subsets of size  $t_t + 1$ . If  $t_t < n/2$ , one of them contains only honest parties, and in that phase, if all parties toss the same coin (we denote it a *lucky* phase), all honest parties obtain the same value.

### 5.3 Protocol

The protocol is very similar to the one in Section 4, but with four changes: 1) it is executed a fixed number of phases, 2) the broadcast protocols are replaced by detectable broadcast protocols (allowing parties for detectable correctness enforcement), 3) the coin is replaced by the one in Section 5.2 and 4) a termination protocol which works even if  $t_t > \max\{t_c, t_v\}$ . In addition, the protocol has a special initial majority round that allows for a simpler analysis of validity and one *lock-round* for each phase that allows the deterministic value of a phase to be fixed before the coins are revealed.

The intuition behind fixing the number of phases is that if the protocol runs indefinitely, it may happen that some messages are *never* scheduled: even though the detectable broadcast *eventually* detects misbehavior, such messages are never scheduled because there are always other messages that the adversary can schedule<sup>1</sup>. However, this cannot happen if the number of phases is fixed. Setting a “large enough” number of phases suffices for parties to reach agreement with probability  $(1 - \epsilon)$ , unless the adversary misbehaved in a detectable broadcast protocol, in which case parties detect it and eventually reach agreement.

We call a batch  $B$  an iteration over all subsets of size  $t_t + 1$ , i.e.  $B = \binom{n}{t_t+1}$ . We set an upper bound  $K + 1$  on the number of batches (hence we have  $(K + 1)B$  phases in total), so that the probability that parties are not in agreement after  $(K + 1)B$  phases is at most  $\epsilon$ .

**Protocol**  $\Pi_{(1-\epsilon)\text{-con}}^{t_c, t_v, t_t}$

*Input.* Every party  $P_i$  holds input  $x_i \in \{0, 1\}$ .

*Variable.*  $y_i = \perp$ .

**Initial majority round** // *This initial round is necessary to ensure validity.*

- Detectable-Broadcast( $x_i$ ) to every party. Wait until  $n - t_t$  messages have been validated.
  - Set  $x_i =$  majority of the  $n - t_t$  validated messages.

**Reaching agreement.** Repeat at most  $K + 1$  times: //  $K + 1$  batches.

- For every subset  $R$  of size  $|R| = t_t + 1$  do: // *we call this loop one batch.*
  1. Detectable-Broadcast( $x_i$ ) to every party. Wait until  $n - t_t$  messages have been validated.

<sup>1</sup> This is the main challenge that one needs to overcome to design an almost-surely terminating consensus for  $t_t \geq n/3$ .

- If all validated messages have the same value  $x$ , update  $x_i = (\text{lock}, x)$ , else update  $x_i = (\text{lock}, ?)$ .
- 2. Detectable-Broadcast( $x_i$ ) to every party. Wait until  $n - t_t$  messages have been validated.
  - If all messages are  $(\text{lock}, x)$  with the same  $x \in \{0, 1\}$ , update  $x_i = (\text{propose}, x)$ , else update  $x_i = (\text{propose}, ?)$ .
- 3. Set  $c_i = \text{Coin}(R)$ .
- 4. Detectable-Broadcast( $x_i$ ) to every party. Wait until  $n - t_t$  messages have been validated.
  - If all messages are  $(\text{propose}, x)$  with  $x \in \{0, 1\}$ , update  $y_i = x$ .
  - Else if at least one of the messages is  $(\text{propose}, x)$  with  $x \in \{0, 1\}$ , then  $x_i = x$ .
  - Otherwise, set  $x_i = c_i$ .

**Termination.**

- Upon updating  $y_i$ , send  $(\text{READY}, y_i)$  to all parties.
- Upon detecting a Byzantine behaviour, send  $(\text{READY}, \perp)$  to all parties.
- Upon receiving  $(\text{READY}, d')$  messages from  $\max\{t_c, t_v\} + 1$  parties that agree on the value  $m' \in \{0, 1, \perp\}$  during the consensus protocol, send  $(\text{READY}, m')$  to all parties.
- Upon receiving  $(\text{READY}, m')$  or  $(\text{TERMINATE})$  from  $n - t_t$  parties from which at least  $\max\{t_v, t_c\} + 1$  are  $\text{READY}$  messages and (they all) agree on the value  $m' \in \{0, 1, \perp\}$ , send  $(\text{TERMINATE})$  to all recipients, output  $m'$  and terminate.

A formal analysis of the protocol can be found in the full version of the paper. Intuitively, if  $f \leq \max\{t_c, t_v\}$ , correctness enforcement ensures the same properties as in the protocol  $\Pi_{\text{as-con}}^{t_c, t_v, t_t}$ , making the proofs of validity and consistency similar to those. However, the termination property involves a bit more careful analysis. The idea is that either each honest party  $P_i$  updates to the same value  $y_i = y$ , or Byzantine behavior is detected. This is because with high probability, there is a phase where honest parties reach agreement by chance (they obtain the same value from the coin, and the coin coincides with the deterministic value of that phase), unless the adversary tampered the outputs from a detectable broadcast protocol in which case it will eventually be detected. As a result, one can argue that there is an honest party that terminates (every honest party eventually sends the same  $\text{READY}$  message), which in turn implies that eventually everyone terminates by a similar argument as for the broadcast protocol in Section 3.

---

**References**

- 1 Marcos K. Aguilera and Sam Toueg. The correctness proof of Ben-Or's randomized consensus algorithm. *Distributed Computing*, 25(5):371–381, 2012. doi:10.1007/s00446-012-0162-z.
- 2 Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In *Proceedings of the Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 27–30. ACM, 1983. doi:10.1145/800221.806707.
- 3 Erica Blum, Jonathan Katz, and Julian Loss. Synchronous consensus with optimal asynchronous fallback guarantees. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019, Part I*, volume 11891 of *LNCS*, pages 131–150. Springer, Heidelberg, December 2019. doi:10.1007/978-3-030-36030-6\_6.
- 4 Erica Blum, Jonathan Katz, and Julian Loss. Network-agnostic state machine replication. Cryptology ePrint Archive, Report 2020/142, 2020. URL: <https://eprint.iacr.org/2020/142>.
- 5 Erica Blum, Chen-Da Liu-Zhang, and Julian Loss. Always have a backup plan: Fully secure synchronous mpc with asynchronous fallback. In Daniele Micciancio and Thomas Ristenpart,


- editors, *Advances in Cryptology - CRYPTO 2020*, pages 707–731, Cham, 2020. Springer International Publishing.
- 6 Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987. doi:10.1016/0890-5401(87)90054-X.
  - 7 Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.
  - 8 Pease, Feldman and Silvio Micali. An optimal probabilistic protocol for synchronous byzantine agreement. *SIAM Journal on Computing*, 26(4):873–933, 1997.
  - 9 Michael J. Fischer, Nancy A. Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. In Michael A. Malcolm and H. Raymond Strong, editors, *4th ACM PODC*, pages 59–70. ACM, August 1985. doi:10.1145/323596.323602.
  - 10 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985. doi:10.1145/3149.214121.
  - 11 Matthias Fitzi, Martin Hirt, Thomas Holenstein, and Jürg Wullschlegler. Two-threshold broadcast and detectable multi-party computation. In Eli Biham, editor, *EUROCRYPT 2003*, volume 2656 of *LNCS*, pages 51–67. Springer, Heidelberg, May 2003. doi:10.1007/3-540-39200-9\_4.
  - 12 Matthias Fitzi, Martin Hirt, and Ueli M. Maurer. Trading correctness for privacy in unconditional multi-party computation (extended abstract). In Hugo Krawczyk, editor, *CRYPTO'98*, volume 1462 of *LNCS*, pages 121–136. Springer, Heidelberg, August 1998. doi:10.1007/BFb0055724.
  - 13 Matthias Fitzi, Thomas Holenstein, and Jürg Wullschlegler. Multi-party computation with hybrid security. In Christian Cachin and Jan Camenisch, editors, *EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 419–438. Springer, Heidelberg, May 2004. doi:10.1007/978-3-540-24676-3\_25.
  - 14 Yue Guo, Rafael Pass, and Elaine Shi. Synchronous, with a chance of partition tolerance. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part I*, volume 11692 of *LNCS*, pages 499–529. Springer, Heidelberg, August 2019. doi:10.1007/978-3-030-26948-7\_18.
  - 15 Martin Hirt, Christoph Lucas, and Ueli Maurer. A dynamic tradeoff between active and passive corruptions in secure multi-party computation. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 203–219. Springer, Heidelberg, August 2013. doi:10.1007/978-3-642-40084-1\_12.
  - 16 Martin Hirt, Christoph Lucas, Ueli Maurer, and Dominik Raub. Graceful degradation in multi-party computation (extended abstract). In Serge Fehr, editor, *ICITS 11*, volume 6673 of *LNCS*, pages 163–180. Springer, Heidelberg, May 2011. doi:10.1007/978-3-642-20728-0\_15.
  - 17 Martin Hirt, Christoph Lucas, Ueli Maurer, and Dominik Raub. Passive corruption in statistical multi-party computation - (extended abstract). In Adam Smith, editor, *ICITS 12*, volume 7412 of *LNCS*, pages 129–146. Springer, Heidelberg, August 2012. doi:10.1007/978-3-642-32284-6\_8.
  - 18 Martin Hirt, Ueli M. Maurer, and Vassilis Zikas. MPC vs. SFE: Unconditional and computational security. In Josef Pieprzyk, editor, *ASIACRYPT 2008*, volume 5350 of *LNCS*, pages 1–18. Springer, Heidelberg, December 2008. doi:10.1007/978-3-540-89255-7\_1.
  - 19 Yuval Ishai, Eyal Kushilevitz, Yehuda Lindell, and Erez Petrank. On combining privacy with guaranteed output delivery in secure multiparty computation. In Cynthia Dwork, editor, *CRYPTO 2006*, volume 4117 of *LNCS*, pages 483–500. Springer, Heidelberg, August 2006. doi:10.1007/11818175\_29.
  - 20 Jonathan Katz. On achieving the “best of both worlds” in secure multiparty computation. In David S. Johnson and Uriel Feige, editors, *39th ACM STOC*, pages 11–20. ACM Press, June 2007. doi:10.1145/1250790.1250793.
  - 21 Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

- 22 Chen-Da Liu-Zhang, Julian Loss, Ueli Maurer, Tal Moran, and Daniel Tschudi. MPC with synchronous security and asynchronous responsiveness. In *Annual International Conference on the Theory and Application of Cryptology and Information Security - ASIACRYPT 2020*, 2020. to appear.
- 23 Julian Loss and Tal Moran. Combining asynchronous and synchronous byzantine agreement: The best of both worlds. *Cryptology ePrint Archive*, Report 2018/235, 2018. URL: <https://eprint.iacr.org/2018/235>.
- 24 Christoph Lucas, Dominik Raub, and Ueli M. Maurer. Hybrid-secure MPC: trading information-theoretic robustness for computational privacy. In Andréa W. Richa and Rachid Guerraoui, editors, *29th ACM PODC*, pages 219–228. ACM, July 2010. doi:10.1145/1835698.1835747.
- 25 Achour Mostéfaoui, Moumen Hamouma, and Michel Raynal. Signature-free asynchronous Byzantine consensus with  $t < n/3$  and  $O(n^2)$  messages. In *ACM Symposium on Principles of Distributed Computing*, pages 2–9. ACM, 2014. doi:10.1145/2611462.2611468.
- 26 Rafael Pass and Elaine Shi. Thunderella: Blockchains with optimistic instant confirmation. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 3–33. Springer, Heidelberg, 2018. doi:10.1007/978-3-319-78375-8\_1.

# Echo-CGC: A Communication-Efficient Byzantine-Tolerant Distributed Machine Learning Algorithm in Single-Hop Radio Network

Qinzi Zhang

Boston College, Chestnut Hill, MA, USA  
zhangbcu@bc.edu

Lewis Tseng 

Boston College, Chestnut Hill, MA, USA  
lewis.tseng@bc.edu

---

## Abstract

In the past few years, many Byzantine-tolerant distributed machine learning (DML) algorithms have been proposed in the point-to-point communication model. In this paper, we focus on a popular DML framework – the parameter server computation paradigm and iterative learning algorithms that proceed in rounds, e.g., [11, 8, 6]. One limitation of prior algorithms in this domain is the *high communication complexity*. All the Byzantine-tolerant DML algorithms that we are aware of need to send  $n$   $d$ -dimensional vectors from worker nodes to the parameter server in each round, where  $n$  is the number of workers and  $d$  is the number of dimensions of the feature space (which may be in the order of millions). In a wireless network, power consumption is proportional to the number of bits transmitted. Consequently, it is extremely difficult, if not impossible, to deploy these algorithms in power-limited wireless devices. Motivated by this observation, we aim to reduce the *communication complexity* of Byzantine-tolerant DML algorithms in the *single-hop radio network* [1, 3, 14].

Inspired by the CGC filter developed by Gupta and Vaidya, PODC 2020 [11], we propose a gradient descent-based algorithm, Echo-CGC. Our main novelty is a mechanism to utilize the *broadcast properties* of the radio network to avoid transmitting the raw gradients (full  $d$ -dimensional vectors). In the radio network, each worker is able to overhear previous gradients that were transmitted to the parameter server. Roughly speaking, in Echo-CGC, if a worker “agrees” with a combination of prior gradients, it will broadcast the “echo message” instead of its raw local gradient. The echo message contains a vector of coefficients (of size at most  $n$ ) and the ratio of the magnitude between two gradients (a float). In comparison, the traditional approaches need to send  $n$  local gradients in each round, where each gradient is typically a vector in a ultra-high dimensional space ( $d \gg n$ ). The improvement on communication complexity of our algorithm depends on multiple factors, including number of nodes, number of faulty workers in an execution, and the cost function. We numerically analyze the improvement, and show that with a large number of nodes, Echo-CGC reduces 80% of the communication under standard assumptions.

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

**Keywords and phrases** Distributed Machine Learning, Single-hop Radio Network, Byzantine Fault, Communication Complexity, Wireless Communication, Parameter Server

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2020.7

**Related Version** A full version of the paper is available at <https://arxiv.org/abs/2011.07447>.

**Acknowledgements** The authors would like to acknowledge Nitin H. Vaidya and anonymous reviewers for their helpful comments.



© Qinzi Zhang and Lewis Tseng;

licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 7; pp. 7:1–7:16

Leibniz International Proceedings in Informatics



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

## 1 Introduction

Machine learning has been widely adopted and explored recently [23, 16]. Due to the exponential growth of datasets and computation power required, distributed machine learning (DML) becomes a necessity. There is also an emerging trend [21, 13] to apply DML in power-limited wireless networked systems, e.g., sensor networks, distributed robots, smart homes, and Industrial Internet-of-Things (IIoT), etc. In these applications, the devices are usually small and fragile, and susceptible to malicious attacks and/or malfunction. More importantly, it is necessary to reduce communication complexity so that (over-)communication does not drain the device battery. Most prior research on fault-tolerant DML (e.g., [8, 4, 11, 6]) has focused on the use cases in clusters or datacenters. These algorithms achieve high resilience (number of faults tolerated), but also incur *high communication complexity*. As a result, most prior Byzantine-tolerant DML algorithms are extremely difficult, if not impossible, to be deployed in power-limited wireless networks.

Motivated by our observations, we aim to design a Byzantine DML algorithm with reduced communication complexity. We consider wireless systems that are modeled as a *single-hop radio network*, and focus on the popular parameter server computation paradigm (e.g., [11, 8, 6]). We propose *Echo-CGC*, and prove its correctness under typical assumptions [4, 8]. For the communication complexity, we formally analyze the expected number of bits that need to be sent from workers to the parameter server. The extension to multi-hop radio network is left as an interesting future work.

**Recent Development in Distributed Machine Learning.** Distributed Machine Learning (DML) is designed to handle a large amount of computation over big data. In the parameter server model, there is a centralized parameter server that distributes the computation tasks to  $n$  workers. These workers have the access to the same dataset (that may be stored externally). Similar to [4, 11, 6], we focus on the *synchronous gradient descent* DML algorithms, where the server and workers proceed in synchronous rounds. In each round, each worker computes a local gradient over the parameter received from the server, and the server then aggregates the gradients collected from workers, and updates the parameter. Under suitable assumptions, prior algorithms [4, 11, 6] converge to the optimal point in the  $d$ -dimensional space  $\mathbb{R}^d$  even if up to  $f$  workers may become Byzantine faulty.

To our knowledge, most Byzantine-tolerant DML or distributed optimization algorithms focused on the case of clusters and datacenters, which are modeled as a point-to-point network. For example, Reference [6], Krum [4], Kardam [7], and ByzSGD [8] focused on the stochastic gradient descent algorithms under several different settings (synchronous, asynchronous, and distributed parameter server). Reference [20, 11, 19] focused on the gradient descent algorithms for the general distributed optimization framework. Zeno [24] uses failure detection to improve the resilience. None of these works aimed to reduce communication complexity.

Another closely related research direction is on reducing the communication complexity of non-Byzantine-tolerant DML algorithms, e.g., [15, 13, 22]. These algorithms are *not* Byzantine fault-tolerant, and adopt a completely different design. For example, reference [15] utilizes relaxed consistency (of the underlying shared data), reference [22] discards coordinates (of the local gradients) aggressively, and reference [13] uses intermediate aggregation. It is not clear how to integrate these techniques with Byzantine fault-tolerance, as these approaches reduce the redundancy, making it difficult to mask the impact from Byzantine workers.

**Single-Hop Radio Network.** We consider the problem in a single-hop radio network, which is a proper theoretical model for wireless networks. Following [1, 3, 14], we assume that single-hop wireless communication is reliable and authenticated, and there is no jamming nor spoofing. Moreover, nodes follow a specific TDMA schedule so that there is no collision. In Section 2.1, we briefly argue why such an assumption is realistic to model wireless communication. In the single-hop radio network model, we aim to minimize the total number of bits to be transmitted in each round. If we directly adapt prior gradient descent-based algorithms [4, 11] to the radio network model, then each worker needs to broadcast a vector of size  $d$ , where  $d$  is the number of dimensions of the feature space. In practical applications (e.g., [9, 13]),  $d$  might be in the order of millions, and the gradients may require a few GBs. Since power consumption is proportional to the communication complexity in wireless channel, prior Byzantine DML algorithms are *not* adequate for power-limited wireless networks..

**Main Contributions.** Inspired by the CGC filter developed by Gupta and Vaidya, PODC 2020 [11], we propose a gradient descent-based algorithm, *Echo-CGC*, for the parameter server model in the single-hop radio network. Our main observation is that since workers can overhear gradients transmitted earlier, they can use this information to avoid sending the raw gradients in some cases. Particularly, if a worker “agrees” with some reference gradient(s) transmitted earlier in the same round, then they send a small message to “echo” with the reference gradient(s). The size of the echo message ( $O(n)$  bits) is negligible compared to the raw gradient ( $O(d)$  bits), since in typical ML applications,  $d \gg n$ .

Our proof is more sophisticated than the one in [11], even though Echo-CGC is inspired by the CGC filter. The reason is that the “echo message” does *not* necessarily contain worker  $i$ ’s local gradient; instead, it can be used to construct an approximate gradient, which intuitively equals a combined gradients between  $i$ ’s local gradients and the gradients broadcast by previous workers. We need to ensure that such an approximation does not affect the aggregation at the server. Moreover, CGC filter [11] works on deterministic gradients – each worker computes the gradient of its local cost function using the full dataset. In our case, each worker computes a stochastic gradient, a gradient over a small random data batch. We prove that with appropriate assumptions, Echo-CGC converges to the optimal point.

Echo-CGC is correct under the same set of assumptions in prior work [4]; however, there is an inherent trade-off between resilience, the proven bound on the communication complexity reduction, and the cost function. Fix the cost function. We derive necessary conditions on  $n$  so that Echo-CGC is guaranteed to perform better. We also perform numerical analysis to understand the trade-off. In general, Echo-CGC saves more and more communication if  $f/n$  becomes smaller and smaller. Moreover, our algorithm performs better when the variance of the data is relatively small. For example, our algorithm tolerates 10% of faulty workers and saves over 75% of communication cost when standard deviation of computed gradients is less than 10% of the true gradient.

## 2 Preliminaries

In this section, we formally define our models, and introduce the assumptions and notations.

### 2.1 Models

**Single-Hop Radio Network.** We consider the standard radio network model in the literature, e.g., [1, 3, 14]. In particular, the underlying communication layer ensures the *reliable local broadcast* property [3]. In other words, the channel is perfectly reliable, and a local broadcast



is correctly received by all neighbors. As noted in [1, 3], this assumption does not typically hold in the current deployed wireless networks, but it is possible to realize such a property with high probability in practice with the help from the MAC layer [2] or physical layer [17].

In our system, nodes can be uniquely identified, i.e., each node has a unique identifier. We assume that a faulty node may not spoof another node’s identity. The communication network is assumed to be single-hop; that is, each pair of nodes are within the communication range of each other. Moreover, time is divided into slots, and each node proceeds synchronously. Message collision is not possible because of the nodes follow a pre-determined TDMA schedule that determine the transmitting node in each slot and the transmission protocol is jam-resistant. Each slot is assumed to be large enough so that it is possible for a node to transmit a gradient. We also assume that each communication round (or communication step) is divided into  $n$  slots, and the TDMA schedule assigns each node to a unique slot. For ease of discussion, node  $i$  is scheduled to transmit at slot  $i$ .

**Stochastic Gradient Descent and Parameter Server.** In this work, we focus on the Byzantine-tolerant distributed Stochastic Gradient Descent (SGD) algorithms, which are popular in the optimization and machine learning literature [4, 8, 11, 5]. Given a cost function  $Q$ , the (sequential) SGD algorithm outputs an optimal parameter  $w^*$  such that

$$w^* = \operatorname{argmin}_{w \in \mathbb{R}^d} Q(w) \quad (1)$$

An SGD algorithm executes in an iterative fashion, where in each round  $t$ , the algorithm computes the gradient of the cost function  $Q$  at parameter  $w^t$  and updates the parameter with the gradient.

*Synchronous Parameter Server Model:* Computation of gradients is typically expensive and slow. One popular framework to speed up the computation is the *parameter server model*, in which the parameter server distributes the computation tasks to  $n$  workers and aggregates their computed gradients to update the parameter in each round. Following the convention, we will use node and worker interchangeably.

We assume a synchronous system, i.e., the computation and communication delays are bounded, and the server and workers know the bound. Consequently, if the server does not receive a message from worker  $i$  by the end of some round, then the server identifies that worker  $i$  is faulty.

Formally speaking, a distributed SGD algorithm in the parameter server model proceeds in synchronous rounds, and executes the following three steps in each round  $t$ :

1. The parameter server broadcasts parameter  $w^t$  to the workers.
2. Each worker  $j$  randomly chooses a random data batch  $\xi_j^t$  from the dataset (shared by all the workers) and computes an estimate,  $g_j^t$ , of the gradient  $\nabla Q(w^t)$  of the cost function  $Q$  using  $\xi_j^t$  and  $w^t$ .
3. The server aggregates estimated gradients from all workers and updates the parameter using the gradient descent approach with step size  $\eta$ :

$$w^{t+1} = w^t - \eta \sum_{j=1}^n g_j^t \quad (2)$$

**Fault Model and Byzantine SGD.** Following [11, 4, 6], our system consists of  $n$  workers, up to  $f$  of which might be Byzantine faulty. We assume that the central parameter server is always fault-free.



Byzantine workers may be controlled by an omniscient adversary which has the knowledge of the current parameter (at the server) and the local gradient of all the other workers, and may have arbitrary behaviors. They *can* send arbitrary messages. However, due to the reliable local broadcast property of the radio network model, they *cannot* send inconsistent messages to the server and other workers. They also *cannot* spoof another node's identity. Our goal is therefore to design a distributed SGD algorithm that solves Equation (1) in the presence of up to  $f$  Byzantine workers.

Workers that are *not* Byzantine faulty are called fault-free workers. These workers follow the algorithm specification faithfully. For a given execution of the algorithm, we denote  $\mathcal{H}$  as the set of fault-free workers and  $\mathcal{B}$  as the set of Byzantine workers. For brevity, we denote  $h = |\mathcal{H}|$  and  $b = |\mathcal{B}|$ ; hence, we have  $b \leq f$  and  $h \geq n - f$ .

**Communication Complexity.** We are interested in minimizing the total number of bits that need to be transmitted from workers to the parameter server in *each round*. Prior algorithms [11, 4] transmit  $n$  gradients in a  $d$ -dimensional space in each round, since each node needs to transmit its local gradient to the centralized server. Typically, each gradient consists of  $d$  floats or doubles (i.e., a single primitive floating point data structure for each dimension).

## 2.2 Assumptions and Notations

We assume that the cost function  $Q$  satisfies some standard properties used in the literature [4, 8, 6], including convexity, differentiability, Lipschitz smoothness, and strong convexity. Following the convention, we use  $\langle a, b \rangle$  to represent the dot product of two vectors  $a$  and  $b$  in the  $d$ -dimensional space  $\mathbb{R}^d$ .

► **Assumption 1** (Convexity and smoothness).  $Q$  is convex and differentiable.

► **Assumption 2** ( $L$ -Lipschitz smoothness). There exists  $L > 0$  such that for all  $w, w' \in \mathbb{R}^d$ ,

$$\|\nabla Q(w) - \nabla Q(w')\| \leq L\|w - w'\| \quad (3)$$

► **Assumption 3** ( $\mu$ -strong convexity). There exists  $\mu > 0$  such that for all  $w, w' \in \mathbb{R}^d$ ,

$$\langle \nabla Q(w) - \nabla Q(w'), w - w' \rangle \geq \mu\|w - w'\|^2 \quad (4)$$

We also assume that the random data batches are independently and identically distributed from the dataset. Before stating the assumptions, we formally introduce the concept of randomness in the framework. Similar to typical stochastic gradient descent algorithms, the only randomness is due to the random data batches  $\xi_j^t$  sampled by each fault-free worker  $j \in \mathcal{H}$  in each round  $t$ , which further makes  $g_j^t$  as well as  $w^{t+1}$  non-deterministic. In the case when a worker uses the entire dataset to train model,  $g_j^t = \nabla Q(w^t)$ . Hence, the result is deterministic, i.e., each fault-free worker derives the same gradient. In practice, data batch is a small sample of the entire data set.<sup>1</sup>

Formally speaking, we denote an operator  $\mathbb{E}_{\Xi^t}(\cdot \mid w^t, \mathcal{G}_{\mathcal{B}}^t)$  as the *conditional expectation* operator over the set of random batches  $\Xi^t = \{\xi_j^t, j = 1, 2, \dots, n\}$  in round  $t$  given (i) the parameter  $w^t$ , and (ii) the set of Byzantine gradients  $\mathcal{G}_{\mathcal{B}}^t = \{g_j^t : j \in \mathcal{B}\}$ . This conditional expectation operator allows us to treat  $w^t$ ,  $Q(w^t)$ , and  $\nabla Q(w^t)$  as constants, as well as the

<sup>1</sup> Reference [11] works on a different formulation in which each worker may have a different local cost function.

Byzantine gradients. This is reasonable because (i) we have the knowledge about  $Q$  and  $w^t$  given an execution, and (ii) the Byzantine gradients are arbitrary, and do not depend on the data batches. From now on, without further specification, we abbreviate the operator  $\mathbb{E}_{\Xi^t}(\cdot | w^t, \mathcal{G}_{\mathcal{B}}^t)$  as  $\mathbb{E}$ .

Below we present two further assumptions of local stochastic gradient  $g_j^t$  at each fault-free worker  $j$ . Similar to [4, 8], we rely on the two following assumptions for correctness proof.

► **Assumption 4** (IID Random Batches). *For all  $j \in \mathcal{H}$  and  $t \in \mathbb{N}$ ,*

$$\mathbb{E}(g_j^t) = \nabla Q(w^t) \quad (5)$$

► **Assumption 5** (Bounded Variance). *For all  $j \in \mathcal{H}$  and  $t \in \mathbb{N}$ ,*

$$\mathbb{E}\|g_j^t - \nabla Q(w^t)\|^2 \leq \sigma^2 \|\nabla Q(w^t)\|^2 \quad (6)$$

**Notation.** We list the most important notations and constants used in our algorithm and analysis in the following table.

■ **Table 1** Notations and constants used in this paper.

$\mathcal{H}$	set of fault-free workers; $h =  \mathcal{H} $
$\mathcal{B}$	set of faulty workers; $b =  \mathcal{B} $
$t$	round number, $t = 0, 1, 2, \dots$
$w^*$	optimal solution to $Q$ , i.e., $w^* = \operatorname{argmin}_{w \in \mathbb{R}^d} Q(w)$
$w^t$	parameter in round $t$
$g_j^t$	estimated gradient of $j$ in round $t$
$\tilde{g}_j^t$	“reconstructed” gradient of $j$ by server in round $t$
$\hat{g}_j^t$	gradient of $j$ in round $t$ after applying the CGC filter
$\eta$	fixed step size as in Equation (2)
$L$	Lipschitz constant
$\mu$	strong convexity constant
$r$	deviation ratio, a key parameter in our algorithm
$k^*$	constant defined in Lemma 2, $k^* \approx 1.12$

### 3 Our Algorithm: Echo-CGC

Our algorithm is inspired by Gupta and Vaidya [11]. Specifically, we integrate their CGC filter with a novel aggregation phase. Our aggregation mechanism utilizes the broadcast property of the radio network to improve the communication complexity. In the CGC algorithm [11], each worker needs to send a  $d$ -dimensional gradient to the server, whereas in our algorithm, some workers only need to send the “echo message” which is of size  $O(n)$  bits. Note that in typical machine learning applications,  $d \gg n$ .

We design our algorithm for the synchronous parameter server model, so the algorithm is presented in an iterative fashion. That is, each worker and the parameter server proceed in synchronous rounds, and the algorithm specifies the exact steps for each round  $t$ . Our Algorithm, Echo-CGC, is presented in Algorithm 1. The algorithm uses the notations and constants summarized in Table 1.

### Algorithm Description

Initially, the parameter server randomly generates an initial parameter  $w^0 \in \mathbb{R}^d$ . Each round  $t \geq 0$  consists of three phases: (i) computation phase, (ii) communication phase, and (iii) aggregation phase. Echo-CGC takes the following inputs: step size  $\eta$ , deviation ratio  $r$ , number of workers  $n$ , and maximum number of tolerable faults  $f$ . The exact requirements on the values of these inputs will become clear later. For example,  $n, f, r$  need to satisfy the bound derived in Lemma 3. More discussion will be presented in Section 4.3.

**Computation Phase.** In the computation phase of round  $t$ , the server broadcasts  $w^t$  to the workers. Each worker  $j$  then computes the local stochastic gradient  $g_j^t = \nabla Q_j(w^t)$  using  $w^t$  and its random data batch  $\xi_j^t$ . Since we assume the parameter server is fault-free, each worker receives the identical  $w^t$ . The local gradient is stochastic, because each worker uses a random data batch to compute the local gradient  $g_j^t$ .

**Communication Phase.** In the communication phase, each worker needs to send the information regarding to its local gradient to the parameter server. This phase is our main novelty, and different from prior algorithms [11, 4, 6]. We utilize the property of the broadcast channel to reduce the communication complexity. As mentioned earlier, the communication phase of round  $t$  is divided into  $n$  slots  $t_1, \dots, t_n$ . Without loss of generality, we assume that each worker  $j$  is scheduled to broadcast its information in slot  $t_j$  (of round  $t$ ). Note that we assume that the underlying physical or MAC layer is jamming-resistant and reliable; hence, each fault-free worker can reliably broadcast the information to all the other nodes.

**Steps for Worker  $j$ .** Each worker  $j$  stores a set of gradients that it overhears in round  $t$ . Denote by  $R_j$  the set of stored gradients. By assumption,  $R_j$  consists of gradients  $g_i^t$  for  $i < j$ , when at the beginning of slot  $t_j$ . Upon receiving a gradient  $g_i^t$  (in the form of a vector in  $\mathbb{R}^d$ ), worker  $j$  stores it to  $R_j$  if  $g_i^t$  is linearly independent with all existing gradients in  $R_j$ . In the slot  $t_j$ , worker  $j$  computes the “echo gradient” using vectors stored in  $R_j$ . Specifically, worker  $j$  takes the following steps:

- It expresses  $R_j$  as  $R_j = \{g_{i_1}^t, \dots, g_{i_{|R_j|}}^t\}$  and constructs a matrix  $A_j \in \mathbb{R}^{d \times |R_j|}$  as

$$A_j^t = \begin{bmatrix} g_{i_1}^t & g_{i_2}^t & \cdots & g_{i_{|R_j|}}^t \end{bmatrix}$$

- It then computes the Moore-Penrose inverse (M-P inverse in short) of  $A_j^t$ , defined as

$$(A_j^t)^+ = ((A_j^t)^T A_j^t)^{-1} (A_j^t)^T,$$

where  $A^T$  is the transpose of matrix  $A$ . The existence of the M-P inverse is guaranteed. Intuitively this is because all columns of  $A_j^t$  are linearly independent by construction. The formal proof is presented in our full paper [25].

- Next, worker  $j$  computes a vector  $x_j^t \in \mathbb{R}^{|R_j|}$  using the M-P inverse:

$$x_j^t = (A_j^t)^+ g_j^t,$$

where  $g_j^t$  is the local stochastic gradient of  $Q$  computed by  $j$  in the computation phase. Note that  $x_j^t$  is of size  $O(n)$ , since  $R_j$  contains at most  $n$  elements.

- Finally, it computes the “echo gradient” as

$$(g_j^t)^* = A_j^t x_j^t$$

Mathematically,  $(g_j^t)^*$  is the projection of  $g_j^t$  onto the span of vectors in  $R_j$ , i.e., the closest vector to  $g_j^t$  in the span of  $R_j$ .

Next, worker  $j$  checks whether the following inequality holds where  $(g_j^t)^*$  is the echo gradient,  $g_j^t$  the local stochastic gradient, and  $r$  the deviation ratio.

$$\|(g_j^t)^* - g_j^t\| \leq r \|g_j^t\| \quad (7)$$

Worker  $j$  performs one of the two actions depending on the result of Inequality (7).

- If Inequality (7) holds, then  $j$  sends the *echo message*  $(\|g_j^t\|/(\|g_j^t\|)^*, x_j^t, I_j^t)$  to the server, where  $I_j^t = \{i_1, \dots, i_{|R_j|}\}$  is a sorted list of worker IDs whose gradients are stored in  $R_j$ .
- Otherwise, worker  $j$  broadcasts the raw gradient  $g_j^t$  to server and all the other workers.

*Steps for Parameter Server:* The parameter server uses a vector  $G$  to store the gradients from workers. Specifically, in each round  $t$ , for each worker  $j$ , the server computes  $\tilde{g}_j^t$  and stores it as the  $j$ -th element of  $G$ . At the beginning of round  $t$ , every element  $G[j]$  is initialized as an empty placeholder  $\perp$ . During the communication phase, the parameter server takes two possible actions upon receiving a message from worker  $j$ :

- If the message is a vector, then the server stores  $\tilde{g}_j^t = g_j^t$  in  $G[j]$ .
- Otherwise, the message is a tuple  $(k, x, I)$ . The server then does the following:
  - If there exists some  $i \in I$  such that  $G[i] = \perp$  (i.e., the server has not received a message from worker  $i$ ), then due to the reliable broadcast property, the server can safely identify  $j$  as a Byzantine worker. By convention, we let the server store  $\tilde{g}_j^t = \vec{0}$ , the zero vector in  $\mathbb{R}^d$ , in  $G[j]$ .
  - Otherwise, denote the matrix  $A_I$  as  $A_I = [G[i_1], \dots, G[i_{|R_j|}]]$  where  $I = \{i_1, \dots, i_{|R_j|}\}$ , and the server stores  $\tilde{g}_j^t$  as  $\tilde{g}_j^t = kA_I x$  in  $G[j]$ .

**Aggregation Phase.** The final phase is identical to the algorithm in [11], in which the server updates the parameter using the CGC filter. First, the server sorts the stored gradients  $G^t$  in the increasing order of their Euclidean norm and relabel the IDs so that  $\|\tilde{g}_{i_1}^t\| \leq \dots \leq \|\tilde{g}_{i_n}^t\|$ . Then the server applies the CGC filter as follows:

$$\hat{g}_j^t = \begin{cases} \frac{\|\tilde{g}_{i_{n-f}}^t\|}{\|\tilde{g}_j^t\|} \tilde{g}_j^t, & j \in \{i_{n-f+1}, \dots, i_n\} \\ \tilde{g}_j^t, & j \in \{i_1, \dots, i_{n-f}\} \end{cases} \quad (8)$$

Finally, the server aggregates the gradients by  $g^t = \sum_{j=1}^n \hat{g}_j^t$  and updates the parameter by  $w^{t+1} = w^t - \eta g^t$ , where  $\eta$  is the fixed step size.

## 4 Convergence Analysis

In this section, we prove the convergence of our algorithm Echo-CGC. The proof is more complicated than the one in [11], even though both algorithms use the CGC filter. This is mainly due to two reasons: (i) we use stochastic gradient, whereas [11] uses a deterministic gradient; and (ii) echo messages only results in an approximate gradient (i.e., the echo gradient which may be deviated from the local stochastic gradient by a ratio  $r$ ). Intuitively, in addition to the Byzantine tampering, we need to deal with non-determinism from stochastic gradients and noise from echo messages.

### 4.1 Convergence Rate Analysis

In this part, we first analyze the *convergence rate*  $\rho$ , which is a constant defined later in Equation (13). Recall a few notations that  $h = |\mathcal{H}|$  and  $b = |\mathcal{B}|$ , where given the execution,  $\mathcal{H}$  is the set of fault-free workers and  $\mathcal{B}$  is the set of Byzantine workers. Also recall that

---

**Algorithm 1** Algorithm Echo-CGC.
 

---

```

1: Parameters:
2:    $\eta > 0$  is the step size defined in Equation (2)
3:    $r > 0$  is the deviation ratio
4:    $n, f, r$  satisfy the resilience bounds stated in Lemma 3
5: Initialization at server:  $w^0 \leftarrow$  a random vector in  $\mathbb{R}^d$ 
6: for  $t \leftarrow 0$  to  $\infty$  do
7:   /* Computation Phase */
8:   At server: broadcast  $w^t$  to all workers;  $G \leftarrow$  a  $\perp$ -vector of length  $n$ 
9:   At worker  $j$ :
10:    receive  $w^t$  from the server
11:     $g_j^t \leftarrow \nabla Q_j(w^t)$ ;  $R_j \leftarrow \{\}$  ▷ local stochastic gradient at worker  $j$ 
12:   /* Communication Phase */
13:   for  $i \leftarrow 1$  to  $n$  do
14:     (i) At worker  $i$ :
15:     if  $|R_i| = 0$  then
16:       broadcast  $g_i^t$ 
17:     else
18:        $A \leftarrow [g]_{g \in R_j}$ ;  $A^+ \leftarrow (A^T A)^{-1} A^T$ ;  $x \leftarrow A^+ g_i^t$  ▷  $Ax$  is the echo gradient
19:       if  $\|Ax - g_i^t\| \leq r \|g_i^t\|$  then
20:          $I \leftarrow \{i' : g_{i'}^t \in R_j\}$  in an ascending order
21:         broadcast  $(\|g_i^t\| / \|Ax\|, x, I)$  ▷ echo message
22:       else
23:         broadcast  $g_i^t$  ▷ raw local gradient
24:       end if
25:     end if
26:     (ii) At worker  $j > i$ :
27:     if  $j$  receives vector  $g_i^t$  from worker  $i$  then
28:        $A \leftarrow [g]_{g \in R_j}$ ;  $A^+ \leftarrow (A^T A)^{-1} A^T$ 
29:       if  $g_i^t$  is linearly independent with  $R_j$  (i.e.,  $AA^+ g_i^t \neq g_i^t$ ) then
30:          $R_i \leftarrow R_i \cup \{g_i^t\}$ 
31:       end if
32:     end if
33:     (iii) At server:
34:     if it receives a vector  $g_j^t$  from worker  $j$  then
35:        $G[j] \leftarrow g_j^t$  ▷  $j$  transmitted a raw gradient
36:     else if it receives an echo message  $(k, x, I)$  from worker  $j$  then
37:       if  $\exists i \in I$  such that  $G[i] = \perp$  then
38:          $G[j] \leftarrow \vec{0}$  ▷  $j$  is a Byzantine worker
39:       else
40:          $A_I \leftarrow [\tilde{g}_i^t]_{i \in I}$ ,  $G[j] \leftarrow k A_I x$  ▷  $j$  transmitted an echo message
41:       end if
42:     end if
43:   end for
44:   /* Aggregation Phase (applying CGC filter from [11]) */
45:    $g^t \leftarrow \sum_{g \in G} CGC(g)$  ▷  $CGC(\cdot)$  defined in Equation (8)
46:    $w^{t+1} \leftarrow w^t - \eta \cdot g^t$  ▷  $\eta$  defined in Equation (2)
47: end for

```

---

## 7:10 Echo-CGC: A Communication-Efficient Byzantine DML

$L$  and  $\mu$  are the constants defined in the Assumption 2 and 3, respectively;  $\sigma$  defined in Assumption 5; and  $r$  is the deviation ratio used in Echo-CGC. To derive  $\rho$ , we need to define series of constants based on the given parameters of  $n, f, h, b, L, \mu, r$ , and  $\sigma$ .

We first define a constant  $\beta$  as

$$\beta = (n - 2f) \frac{\mu - r(1 + \sigma)L}{1 + r} - b(1 + k_h \sigma)L, \quad (9)$$

where  $k_x$  is defined as

$$k_x = 1 + \frac{x - 1}{\sqrt{2x - 1}}, \quad \forall x \geq 1. \quad (10)$$

We then define a constant  $\gamma$  as

$$\gamma = nL^2 (h(1 + \sigma^2) + b\alpha_h), \quad (11)$$

where

$$\alpha_x = x\sigma^2 + (1 + k_h \sigma)^2, \quad \forall x \geq 1. \quad (12)$$

Finally, we define the convergence rate  $\rho$  using  $\beta$  and  $\gamma$  as follows:

$$\rho = 1 - 2\beta\eta + \gamma\eta^2. \quad (13)$$

We will prove that under some standard assumptions, the convergence rate  $\rho$  is in the interval  $[0, 1)$ . We first present several auxiliary lemmas. Due to page limit, most proofs are presented in the full paper [25].

► **Lemma 1.** *Let  $L, \mu > 0$  be the Lipschitz constant and strong convexity constant defined in Assumption 2 and 3, respectively. Then we have  $\mu \leq L$ .*

► **Lemma 2.** *Denote  $k^* = \sup_x \{k_x / \sqrt{x} : x \geq 1\}$ . Then  $k^* < \infty$ , and numerically  $k^* \approx 1.12$ . Equivalently,  $k_h \leq k^* \sqrt{h}$  for all  $h \geq 1$ .*

► **Lemma 3.** *Assume  $n\mu - (3 + k_n \sigma)fL > 0$ , then there exists  $r > 0$  that satisfies equation below.*

$$r < \frac{n\mu - (3 + k_n \sigma)fL}{(n - 2f)(1 + \sigma)L + (1 + k_n \sigma)fL}. \quad (14)$$

Moreover, if  $r > 0$  satisfies Equation (14), then  $\beta > 0$ .

Lemma 3 implies that we need to bound  $\sigma$  for convergence. In general, Echo-CGC is correct if  $\sigma = o(\log n)$ . For brevity, we make the following assumption to simplify the proof of convergence and the analysis of communication complexity. We stress that this assumption can be relaxed using basically the same analysis with a denser mathematical manipulation.

► **Assumption 6.** *Let  $\sigma$  be the variance bound defined in Assumption 5. We further assume that  $\sigma < \frac{1}{\sqrt{n}}$ .*

Under Assumption 6, we can narrow down the bound of  $r$  in Lemma 3 to loosen our assumption on fault tolerance.

► **Lemma 4.** *Assume  $n\mu - (3 + k^*)fL > 0$  ( $k^* \approx 1.12$ ), then there exists  $r > 0$  satisfying Equation (15) such that  $\beta > 0$ .*

$$r < \frac{n\mu - (3 + k^*)fL}{(n - 2f)(1 + \sigma)L + (1 + k^*)fL}. \quad (15)$$

► **Theorem 5.** *Assume  $n\mu - (3 + k^*)fL > 0$  and  $r$  is a value that satisfies Inequality (15). Then we can find an  $\eta > 0$  such that  $\eta < 2\beta/\gamma$ , which in turn makes  $\rho \in [0, 1)$ .*

## 4.2 Proof of Convergence

Next, we prove the convergence of our algorithm. That is, Echo-CGC converges to the optimal point  $w^*$  of the cost function  $Q$ . We prove the convergence under the assumption that  $n\mu - (3 + k^*)fL > 0$ . Due to page limit, we present key proofs here, and the rest can be found in [25].

Recall our definition of the conditional expectation  $\mathbb{E} = \mathbb{E}_{\Xi^t}(\cdot \mid w^t, G_B^t)$  introduced in Section 2.2. Before proving the main theorem, we introduce some preliminary lemmas.

► **Lemma 6.** *For all  $t$  and for all  $j \in \mathcal{H}$ ,*

$$\mathbb{E}\|g_j^t\| \leq (1 + \sigma)\|\nabla Q(w^t)\|. \quad (16)$$

► **Lemma 7.** *Recall that  $\hat{g}_j^t$  is the gradient after applying the CGC filter. For all  $t$  and for all  $j \in \{1, 2, \dots, n\}$ ,*

$$\mathbb{E}\|\hat{g}_j^t\| \leq (1 + k_h\sigma)\|\nabla Q(w^t)\|. \quad (17)$$

The proof of Lemma 7 is based on Lemma 6 and the following prior results: Gumbel [10] and Hartley and David [12] proved that given identical means and variances  $(\mu, \sigma^2)$ , the upper bound of the expectation of the largest random variable among  $n$  independent random variables is  $\mu + \frac{\sigma(n-1)}{\sqrt{2n-1}}$ .

► **Lemma 8.** *Following the same setup, for all  $t$  and for all  $j \in \{1, 2, \dots, n\}$ ,*

$$\mathbb{E}\|\hat{g}_j^t\|^2 \leq \alpha_h\|\nabla Q(w^t)\|^2. \quad (18)$$

The proof of Lemma 8 is based on Lemma 6 and the following result: Papadatos [18] proved that for  $n$  i.i.d. random variables  $X_1 \leq X_2 \leq \dots \leq X_n$  with finite variance  $\sigma^2$ , the maximum variance of  $X_n$  is bounded above by  $n\sigma^2$ .

Lemma 7 and Lemma 8 provide upper bounds on  $\mathbb{E}\|\hat{g}_j^t\|$  and  $\mathbb{E}\|\hat{g}_j^t\|^2$ . These two bounds allow us to bound the impact of bogus gradients transmitted by a faulty node  $j$ . If  $j$  transmitted an extreme gradient, it would be dropped by the CGC filter; otherwise, these two bounds essentially imply that the filtered gradient  $\hat{g}_j^t$  has some nice property even if  $j$  is faulty. For fault-free gradients, Lemma 6 provides a better bound.

► **Theorem 9.** *Assume that  $n\mu - (3 + k^*)fL > 0$ . We can find  $r > 0$  that satisfies Inequality (15) and  $\eta > 0$  such that  $\eta < 2\beta/\gamma$ . Echo-CGC with the chosen  $r$  and  $\eta$  will converge to the optimal parameter  $w^*$  as  $t \rightarrow \infty$ .*

**Proof.** Our ultimate goal is to show that the sequence  $\{\mathbb{E}\|w^t - w^*\|^2\}_{t=0}^\infty$  converges to 0. Recall that the aggregation rule of the algorithm is  $w^{t+1} = w^t - \eta g^t$ . Thus, we obtain that

$$\begin{aligned} \mathbb{E}\|w^{t+1} - w^*\|^2 &\leq \mathbb{E}\|w^t - w^* - \eta g^t\|^2 \\ &= \underbrace{\mathbb{E}\|w^t - w^*\|^2}_A - \underbrace{2\eta\mathbb{E}\langle w^t - w^*, g^t \rangle}_B + \underbrace{\eta^2\mathbb{E}\|g^t\|^2}_C. \end{aligned} \quad (19)$$

Since  $w^t$  is known,  $w^t$  can be treated as a constant, and  $\mathbb{E}\|w^t - w^*\|^2 = \|w^t - w^*\|^2$ .

**Part C:** In [25], we show that the following inequality holds.

$$\mathbb{E}\|g^t\|^2 \leq \gamma\|w^t - w^*\|^2. \quad (20)$$

**Part B:** By linearity of inner product,

$$\langle w^t - w^*, g^t \rangle = \sum_{j \in \mathcal{H}} \langle w^t - w^*, \hat{g}_j^t \rangle + \sum_{j \in \mathcal{B}} \langle w^t - w^*, \tilde{g}_j^t \rangle. \quad (21)$$

First, by Schwarz Inequality,  $\langle w^t - w^*, \hat{g}_j^t \rangle \geq -\|w^t - w^*\| \|\hat{g}_j^t\|$ ; by Lemma 7 and  $L$ -Lipschitz assumption,  $\mathbb{E} \|\hat{g}_j^t\| \leq (1 + k_h \sigma) L \|w^t - w^*\|$ . Thus,

$$\mathbb{E} \langle w^t - w^*, \hat{g}_j^t \rangle \geq -(1 + k_h \sigma) L \|w^t - w^*\|^2, \quad \forall j \in \mathcal{B}. \quad (22)$$

Next, observe that by our algorithm, for each  $j \in \mathcal{H}$ , the received gradient before CGC filter  $\tilde{g}_j^t$  satisfies (i)  $\|\tilde{g}_j^t\| = \|g_j^t\|$  and (ii)  $\tilde{g}_j^t = a_j(g_j^t + \Delta g_j^t)$ , for some constant  $a_j = \|g_j^t\| / \|g_j^t + \Delta g_j^t\|$  and a vector  $\Delta g_j^t$  such that  $\|\Delta g_j^t\| \leq r \|g_j^t\|$ . This implies  $a_j \geq 1/(1+r)$ . Therefore,

$$\begin{aligned} \mathbb{E} \langle w^t - w^*, \tilde{g}_j^t \rangle &= \mathbb{E} \langle w^t - w^*, a_j(g_j^t + \Delta g_j^t) \rangle \\ &\geq \frac{1}{1+r} (\langle w^t - w^*, \mathbb{E} g_j^t \rangle + \mathbb{E} \langle w^t - w^*, \Delta g_j^t \rangle), \quad \forall j \in \mathcal{H}. \end{aligned} \quad (23)$$

By Assumption 4,  $\mathbb{E} g_j^t = \nabla Q(w^t)$ ; by strong convexity,

$$\langle w^t - w^*, \nabla Q(w^t) \rangle \geq \mu \|w^t - w^*\|^2.$$

By Schwarz inequality,  $\mathbb{E} \langle w^t - w^*, \Delta g_j^t \rangle \geq -\|w^t - w^*\| \mathbb{E} \|\Delta g_j^t\|$ ; and  $\mathbb{E} \|\Delta g_j^t\| \leq r \mathbb{E} \|g_j^t\|$ . By Lemma 6 and  $L$ -Lipschitz assumption,  $\mathbb{E} \|g_j^t\| \leq (1 + \sigma) L \|w^t - w^*\|$ . Thus,

$$\mathbb{E} \langle w^t - w^*, \Delta g_j^t \rangle \geq -r(1 + \sigma) L \|w^t - w^*\|^2.$$

Upon substituting these results into Equation (23), we obtain that

$$\mathbb{E} \langle w^t - w^*, \tilde{g}_j^t \rangle \geq \frac{\mu - r(1 + \sigma)L}{1 + r} \|w^t - w^*\|^2, \quad \forall j \in \mathcal{H}. \quad (24)$$

We partition  $\mathcal{H}$  into two parts:  $\mathcal{H}_1 = \mathcal{H} \cap \{i_1, \dots, i_{n-f}\}$  and  $\mathcal{H}_2 = \mathcal{H} \setminus \mathcal{H}_1$ . For each  $j \in \mathcal{H}_1$ , the received gradient is unchanged by CGC filter, i.e.,  $\hat{g}_j^t = \tilde{g}_j^t$ . Therefore, Equation (24) also holds for  $\hat{g}_j^t$ , for all  $j \in \mathcal{H}_1$ .

The case of  $\mathcal{H}_2$  is similar. Note that for each  $j \in \mathcal{H}_2$ , the gradient  $\tilde{g}_j^t$  is scaled down to  $\hat{g}_j^t$  by CGC filter. In other words, there exists some constant  $a'_j \geq 0$  such that  $\hat{g}_j^t = a'_j \tilde{g}_j^t$ . Therefore, by Equation (23),

$$\mathbb{E} \langle w^t - w^*, \hat{g}_j^t \rangle = \mathbb{E} \langle w^t - w^*, a'_j \tilde{g}_j^t \rangle = a'_j \mathbb{E} \langle w^t - w^*, \tilde{g}_j^t \rangle, \quad \forall j \in \mathcal{H}_2.$$

We can verify that if by assumption that  $r > 0$  satisfies Equation (15), then  $\mu - r(1 + \sigma)L > 0$ ; and Equation (23) implies that  $\mathbb{E} \langle w^t - w^*, \tilde{g}_j^t \rangle \geq 0$ . Therefore,

$$\mathbb{E} \langle w^t - w^*, \hat{g}_j^t \rangle \geq 0, \quad \forall j \in \mathcal{H}_2. \quad (25)$$

Note that  $|\mathcal{H}_1| \geq h - 2f$ . Upon substituting Equation (22), (24), (25) into Equation (21), we obtain that

$$\mathbb{E} \langle w^t - w^*, g^t \rangle \geq \left( (n - 2f) \frac{\mu - r(1 + \sigma)L}{1 + r} - b(1 + k_h \sigma)L \right) \|w^t - w^*\|^2. \quad (26)$$

By definition of  $\beta$  in Equation (9), this implies  $\mathbb{E} \langle w^t - w^*, g^t \rangle \geq \beta \|w^t - w^*\|^2$ .

**Conclusion:** Upon combining part A, B and C, by definition of  $\rho$  in Equation (13),

$$\mathbb{E} \|w^{t+1} - w^*\|^2 \leq \rho \|w^t - w^*\|^2, \quad \forall t = 0, 1, 2, \dots$$

Recall the definition of the conditional expectation operator  $\mathbb{E}$ . This implies that

$$\mathbb{E} (\|w^t - w^*\|^2 \mid w^0, \mathcal{G}_{\mathcal{B}}^0, \dots, \mathcal{G}_{\mathcal{B}}^t) \leq \rho^t \|w^0 - w^*\|^2$$

By Theorem 5,  $\rho \in (0, 1)$ . Therefore, as  $t \rightarrow \infty$ ,  $\|w^t - w^*\|^2$  converges to 0. In other words,  $w^t$  converges to the optimal parameter  $w^*$ . This proves the theorem.  $\blacktriangleleft$



### 4.3 Communication Complexity

We analyze the communication complexity of the Echo-CGC algorithm, and show that under suitable conditions, it effectively reduces communication complexity compared to prior algorithms [4, 11]. First consider a ball in  $\mathbb{R}^d$  whose center is the true gradient  $\nabla Q(w^t)$ :

$$B(\nabla Q(w^t), \frac{r}{2+r} \|\nabla Q(w^t)\|) = \{u \in \mathbb{R}^d : \|u - \nabla Q(w^t)\| \leq \frac{r}{2+r} \|\nabla Q(w^t)\|\}, \quad (27)$$

where  $r > 0$  is the deviation ratio. For a slight abuse of notations, we abbreviate the ball as  $B$ . This should not be confused with  $\mathcal{B}$ , the set of Byzantine workers. We present only the main results, and the proofs can be found in [25].

► **Lemma 10.** *For all  $u, v \in B$ ,  $\|u - v\| \leq r\|u\|$  (and  $\|u - v\| \leq r\|v\|$ ).*

Given Lemma 10, we compute the probability that an arbitrary gradient  $g_j^t$  is in the ball  $B$ . By Markov's Inequality,

$$\begin{aligned} \Pr(g_j^t \in B) &= \Pr\left(\|g_j^t - \nabla Q(w^t)\|^2 \leq \frac{r^2}{(2+r)^2} \|\nabla Q(w^t)\|^2\right) \\ &\geq 1 - \frac{\mathbb{E}\|g_j^t - \nabla Q(w^t)\|^2}{\frac{r^2}{(2+r)^2} \|\nabla Q(w^t)\|^2}. \end{aligned} \quad (28)$$

By Assumption 5,  $\mathbb{E}\|g_j^t - \nabla Q(w^t)\|^2 \leq \sigma^2 \|\nabla Q(w^t)\|^2$ , so we conclude that  $\Pr(g_j^t \in B) \geq p$ , where  $p$  is the lower bound defined as  $p = 1 - (1 + 2/r)^2 \sigma^2$ .

Denote  $n_B = |\{j : g_j^t \in B\}|$  and  $n^*$  as the number of workers that send the ‘‘echo message’’ in a round. By Lemma 10,  $n^* \geq n_B - 1$ . Since each event  $\{g_j^t \in B\}$  is independent and has a fixed probability,  $n^*$  follows a Binomial distribution with success probability  $\Pr(g_j^t \in B)$  which is bounded below by  $p$ . Therefore,

$$\mathbb{E}n^* \geq \mathbb{E}n_B - 1 \geq np - 1.$$

For  $n \gg 1$ , we assume that  $1/n \approx 0$ . Also in practice,  $d \gg n$ , so the message complexity of each echo message (in  $O(n)$  bits) is negligible compared to raw gradients (in  $O(d)$  bits). Hence, the ratio of bit complexity of our algorithm and prior algorithms (e.g., [4, 11]) can be approximately bounded above as follows:

$$\begin{aligned} \frac{\text{bit complexity of Echo-CGC}}{\text{bit complexity of prior algorithms}} &= \frac{n^*O(n) + (n - n^*)O(d)}{nO(d)} \\ &\leq \frac{(np - 1)O(n) + [n - (np - 1)]O(d)}{nO(d)} \\ &\approx 1 - p. \end{aligned}$$

We denote the upper bound of ratio of reduced complexity to complexity of prior algorithms as  $C = 1 - p = (1 + 2/r)^2 \sigma^2$ .

**Analysis.** By Equation (3) and Lemma 2,  $C$  can be expressed as

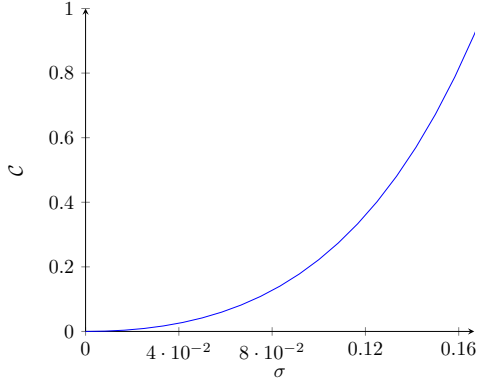
$$C \leq \sigma^2 \left(1 + 2 \cdot \frac{(1 - 2x)(1 + \sigma) + (1 + \sigma k^* \sqrt{n})x}{\mu/L - (3 + \sigma k^* \sqrt{n})x}\right)^2, \quad (29)$$

where  $x = f/n$  is the fault-tolerance factor.

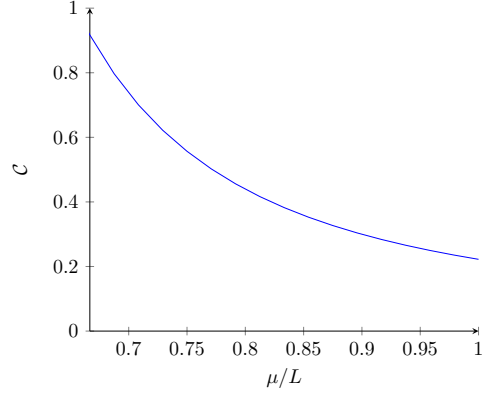
As Equation (29) shows, the ratio  $C$  is related to four non-trivial variables: (i) bound of variance  $\sigma \geq 0$ ; (ii) resilience  $x = f/n$  satisfying the assumption in Lemma 3, i.e.,

$$\mu/L - (3 + \sigma k^* \sqrt{n})x > 0;$$

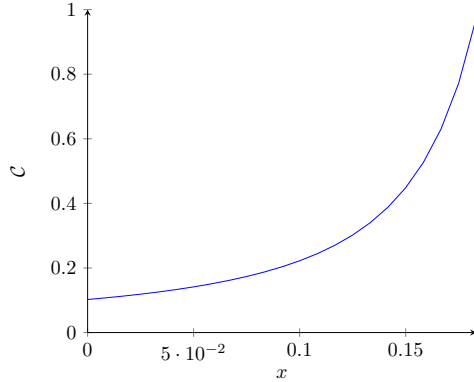
(iii) constant  $L/\mu$ , which is determined by the cost function  $Q$  and satisfies  $0 < L/\mu < 1$  by Lemma 1; and (iv) number of workers  $n > 0$ .



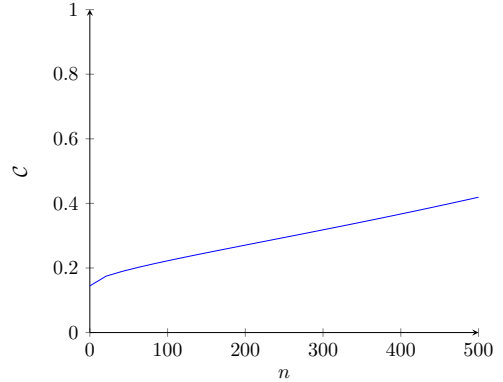
(a)  $C$  as a function of  $\sigma$ , for fixed  $\mu/L = 1$ ,  $x = 0.1$ , and  $n = 100$ .



(b)  $C$  as a function of  $\mu/L$ , for fixed  $\sigma = 0.1$ ,  $x = 0.1$ , and  $n = 100$ .



(c)  $C$  as a function of  $x$ , for fixed  $\sigma = 0.1$ ,  $\mu/L = 1$ , and  $n = 100$ .



(d)  $C$  as a function of  $n$ , for fixed  $\sigma = 0.1$ ,  $\mu/L = 1$ , and  $x = 0.1$ .

We first plot the relation between one factor and  $C$  while fixing the other three factors. First, we present the most significant fact,  $\sigma$ . We fix  $\mu/L = 1$ ,  $x = 0.1$ , and  $n = 100$ . As Figure 1a shows,  $C$  increases in an almost quadratic speed with  $\sigma$  because of the  $\sigma^2$  term in Equation (29). Therefore, our algorithm is guaranteed to have lower communication complexity when the variance of gradients is relatively low, especially when  $\sigma \leq 0.1$ . In practice, this is the scenario when the data set consists mainly of similar data instances.

Then, we plot  $C$  against  $\mu/L$  with fixed  $\sigma = 0.1$ ,  $x = 0.1$ , and  $n = 100$ . As Figure 1b shows,  $C$  decreases as  $\mu/L$  becomes closer to 1. As  $\mu/L > 0.75$ ,  $C < 0.5$ , meaning that  $[0.75, 1]$  is the range of  $\mu/L$  where our algorithm is guaranteed to perform significantly better.

Next, we plot  $C$  against  $x$  with fixed  $\sigma = 0.1$ ,  $\mu/L$ , and  $n = 100$ . As Figure 1c shows, there is a trade-off between  $C$  and fault resilience  $x$ . As  $x$  approaches the max resilience defined in Lemma 3, i.e.,  $x_{\max} = \frac{\mu/L}{(3 + \sigma k^* \sqrt{n})}$ , the theoretical upper bound  $C$  blows up. Moreover, as  $x < 0.15$ ,  $C < 0.4$ ; and thus  $[0, 0.15]$  is a proper range of  $x$ .

Finally, we plot  $C$  against  $n$  with fixed  $\sigma = 0.1$ ,  $\mu/L = 1$ , and  $x = 0.1$ . As Figure 1d shows,  $C$  increases almost linearly with respect to  $n$  with a relatively flat slope. In other words,  $n$  is *not* a significant factor of  $C$ ; and the performance of our algorithm is stable in a wide range of  $n$ .

In conclusion, our algorithm is guaranteed to require lower communication complexity when: (i)  $\sigma$  is low, i.e., data instances are similar and (ii)  $\mu/L$  is close to 1. Also, there is a trade-off between resilience and efficiency. As a concrete example, when  $\sigma = 0.1$ ,  $x = 0.2$ ,  $\mu/L = 1$ , and  $n = 100$ ,  $C \approx 0.25$ , meaning that our algorithm is guaranteed to save at least 75% of communication cost.

## 5 Summary

In this paper, we present our Byzantine-tolerant DML algorithm that incurs lower communication complexity in a single-hop radio network (under suitable conditions). Our algorithm is inspired by the CGC filter [11], but we need to devise new proofs to handle the randomness and noise introduced in our mechanism.

There are two interesting open problems: (i) multi-hop radio network; and (ii) different mechanism for constructing echo messages, e.g., usage of angles rather than distance ratio.

---

## References

- 1 Dan Alistarh, Seth Gilbert, Rachid Guerraoui, Zarko Milosevic, and Calvin Newport. Securing every bit: Authenticated broadcast in radio networks. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, page 50–59, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1810479.1810489.
- 2 Baruch Awerbuch, Andrea Richa, and Christian Scheideler. A jamming-resistant mac protocol for single-hop wireless networks. In *Proceedings of the Twenty-Seventh ACM Symposium on Principles of Distributed Computing*, PODC '08, page 45–54, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1400751.1400759.
- 3 Vartika Bhandari and Nitin H. Vaidya. On reliable broadcast in a radio network. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing*, PODC '05, page 138–147, New York, NY, USA, 2005. Association for Computing Machinery. doi:10.1145/1073814.1073841.
- 4 Peva Blanchard, El Mahdi El Mhamdi, Rachid Guerraoui, and Julien Stainer. Machine learning with adversaries: Byzantine tolerant gradient descent. In *NIPS'17*, page 118–128, Red Hook, NY, USA, 2017. Curran Associates Inc.
- 5 Stephen Boyd and Lieven Vandenbergh. *Convex Optimization*. Cambridge University Press, USA, 2004.
- 6 Yudong Chen, Lili Su, and Jiaming Xu. Distributed statistical machine learning in adversarial settings: Byzantine gradient descent. *Proc. ACM Meas. Anal. Comput. Syst.*, 1(2), December 2017. doi:10.1145/3154503.
- 7 Georgios Damaskinos, El Mahdi El Mhamdi, Rachid Guerraoui, Richeek Patra, and Mahsa Taziki. Asynchronous Byzantine machine learning (the case of SGD). In Jennifer Dy and Andreas Krause, editors, *Proceedings of Machine Learning Research*, volume 80, pages 1145–1154, Stockholm, Sweden, 2018. PMLR. URL: <http://proceedings.mlr.press/v80/damaskinos18a.html>.
- 8 El-Mahdi El-Mhamdi, Rachid Guerraoui, Arsany Guirguis, Lê Nguyễn Hoàng, and Sébastien Rouault. Genuinely distributed byzantine machine learning. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, PODC '20, page 355–364, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3382734.3405695.
- 9 J. Fan and J. Lv. A selective overview of variable selection in high dimensional feature space. *Statistica Sinica*, pages 101–148, January 2010.

- 10 E. J. Gumbel. The maxima of the mean largest value and of the range. *The Annals of Mathematical Statistics*, 25(1):76–84, 1954. URL: <http://www.jstor.org/stable/2236513>.
- 11 Nirupam Gupta and Nitin H. Vaidya. Fault-tolerance in distributed optimization: The case of redundancy. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, PODC '20, page 365–374, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3382734.3405748.
- 12 H. O. Hartley and H. A. David. Universal bounds for mean range and extreme observation. *The Annals of Mathematical Statistics*, 25(1):85–99, 1954. URL: <http://www.jstor.org/stable/2236514>.
- 13 Seyyedali Hosseinalipour, Christopher G. Brinton, Vaneet Aggarwal, Huaiyu Dai, and Mung Chiang. From federated learning to fog learning: Towards large-scale distributed machine learning in heterogeneous wireless networks, 2020. arXiv:2006.03594.
- 14 Chiu-Yuen Koo. Broadcast in radio networks tolerating byzantine adversarial behavior. In *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing*, PODC '04, page 275–282, New York, NY, USA, 2004. Association for Computing Machinery. doi:10.1145/1011767.1011807.
- 15 Mu Li, David G. Andersen, Alexander Smola, and Kai Yu. Communication efficient distributed machine learning with the parameter server. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'14, page 19–27, Cambridge, MA, USA, 2014. MIT Press.
- 16 Ruben Mayer and Hans-Arno Jacobsen. Scalable deep learning on distributed infrastructures: Challenges, techniques, and tools. *ACM Comput. Surv.*, 53(1), February 2020. doi:10.1145/3363554.
- 17 V. Navda, A. Bohra, S. Ganguly, and D. Rubenstein. Using channel hopping to increase 802.11 resilience to jamming attacks. In *IEEE INFOCOM 2007 - 26th IEEE International Conference on Computer Communications*, pages 2526–2530, 2007.
- 18 Nickos Papadatos. Maximum variance of order statistics. *Annals of the Institute of Statistical Mathematics*, 47:185–193, February 1995. doi:10.1007/BF00773423.
- 19 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.
- 20 Lili Su and Nitin H. Vaidya. Non-bayesian learning in the presence of byzantine agents. In *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings*, pages 414–427, 2016. doi:10.1007/978-3-662-53426-7\_30.
- 21 Y. Sun, M. Peng, Y. Zhou, Y. Huang, and S. Mao. Application of machine learning in wireless networks: Key techniques and open issues. *IEEE Communications Surveys Tutorials*, 21(4):3072–3108, 2019.
- 22 Zeyi Tao and Qun Li. esgd: Communication efficient distributed deep learning on the edge. In *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*, Boston, MA, July 2018. USENIX Association. URL: <https://www.usenix.org/conference/hotedge18/presentation/tao>.
- 23 Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg, Tim Verbelen, and Jan S. Rellermeier. A survey on distributed machine learning. *ACM Comput. Surv.*, 53(2), March 2020. doi:10.1145/3377454.
- 24 Cong Xie, Sanmi Koyejo, and Indranil Gupta. Zeno: Distributed stochastic gradient descent with suspicion-based fault-tolerance. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of Machine Learning Research*, volume 97, pages 6893–6901, Long Beach, California, USA, 2019. PMLR. URL: <http://proceedings.mlr.press/v97/xie19b.html>.
- 25 Qinzi Zhang and Lewis Tseng. Echo-CGC: A communication-efficient byzantine-tolerant distributed machine learning algorithm in single-hop radio network, 2020. arXiv:2011.07447.

# AKSEL: Fast Byzantine SGD

**Amine Boussetta**<sup>1</sup>

Mohammed VI Polytechnic University, Ben Guerir, Morocco  
amine.boussetta@um6p.ma

**El-Mahdi El-Mhamdi**

EPFL, Lausanne, Switzerland  
elmahdielmhamdi@gmail.com

**Rachid Guerraoui**

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

**Alexandre Maurer**

Mohammed VI Polytechnic University, Ben Guerir, Morocco  
alexandre.maurer@um6p.ma

**Sébastien Rouault**

EPFL, Lausanne, Switzerland  
sebastien.rouault@epfl.ch

---

## Abstract

Modern machine learning architectures distinguish servers and workers. Typically, a  $d$ -dimensional model is hosted by a server and trained by  $n$  workers, using a distributed *stochastic gradient descent* (SGD) optimization scheme. At each SGD step, the goal is to estimate the gradient of a cost function. The simplest way to do this is to *average* the gradients estimated by the workers. However, averaging is not resilient to even one single Byzantine failure of a worker. Many alternative *gradient aggregation rules* (GARs) have recently been proposed to tolerate a maximum number  $f$  of Byzantine workers. These GARs differ according to (1) the complexity of their computation time, (2) the maximal number of Byzantine workers despite which convergence can still be ensured (breakdown point), and (3) their accuracy, which can be captured by (3.1) their angular error, namely the angle with the true gradient, as well as (3.2) their ability to aggregate full gradients. In particular, many are not *full gradients* for they operate on each dimension separately, which results in a coordinate-wise blended gradient, leading to low accuracy in practical situations where the number ( $s$ ) of workers that are actually Byzantine in an execution is small ( $s \ll f$ ).

We propose AKSEL, a new scalable median-based GAR with optimal time complexity ( $\mathcal{O}(nd)$ ), optimal breakdown point ( $n > 2f$ ) and the lowest upper bound on the *expected angular error* ( $\mathcal{O}(\sqrt{d})$ ) among *full gradient* approaches. We also study the *actual angular error* of AKSEL when the gradient distribution is normal and show that it only grows in  $\mathcal{O}(\sqrt{d} \log n)$ , which is the first logarithmic upper bound ever proven on the number of workers  $n$  assuming an optimal breakdown point. We also report on an empirical evaluation of AKSEL on various classification tasks, which we compare to alternative GARs against state-of-the-art attacks. AKSEL is the only GAR reaching top accuracy when there is actually none or few Byzantine workers while maintaining a good defense even under the extreme case ( $s = f$ ). For simplicity of presentation, we consider a scheme with a single server. However, as we explain in the paper, AKSEL can also easily be adapted to multi-server architectures that tolerate the Byzantine behavior of a fraction of the servers.

**2012 ACM Subject Classification** Computing methodologies → Batch learning; Security and privacy → Distributed systems security; Theory of computation → Nonconvex optimization

**Keywords and phrases** Machine learning, Stochastic gradient descent, Byzantine failures

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2020.8

---

<sup>1</sup> Corresponding author



© Amine Boussetta, El-Mahdi El-Mhamdi, Rachid Guerraoui, Alexandre Maurer, and Sébastien Rouault;

licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 8; pp. 8:1–8:16



Leibniz International Proceedings in Informatics

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

## 1 Introduction

Machine learning (ML) has gained a lot of attention during the last decades, where data collection and processing have reached outstanding levels in terms of volume, variety and velocity. Public awareness of machine learning, especially after the renaissance of neural networks with the backpropagation algorithm [16], increased greatly when companies like IBM and DeepMind created computer programs that beat world class champions in various games. Machine learning started being incorporated within many applications such as transportation, healthcare, finance, agriculture, retail, and customer service.

Essentially, training a supervised ML algorithm consists in determining the set of parameters that minimize the error between the model prediction and the actual output, a scheme formally called *empirical risk minimization* [27]. In a single machine, it is common to use *Gradient Descent* (GD) to minimize the cost function (which depends on the entire dataset) by computing its gradient. For modern applications however, even the best and most expensive hardware would eventually become insufficient.

Almost every industry grade machine learning algorithm is nowadays implemented in a distributed manner. Most rely on *stochastic gradient descent* (SGD) [25], a variant of GD that supports parallelization. However, a distributed architecture induces many challenges, in particular the risk of partial failures. The classical way to model various failures (e.g. software bug, arbitrary behavior of the hardware. . .) is the *Byzantine* abstraction and the classical way to deal with them is to use a state machine replication protocol [26], but this solution entails heavy communication and computational costs.

More specifically, distributed implementations of SGD typically consist of parameter servers and workers. For simplicity of presentation, we consider the now classical ML scheme with a single parameter server and several workers [1] (but our result can easily be extended to a setting with multiple servers). The dataset is distributed over these workers, each of which computes an estimation of the gradient step based on their share of the data. The parameter server aggregates all the received gradient estimations and updates the parameter vector accordingly. The goal is to come up with an estimate of the (true) gradient that would have been computed on a single machine using GD. The simplest and best way to aggregate the vectors is through averaging [23] which comes very close to the true gradient. However, averaging cannot withstand a single Byzantine failure of a worker [4].

To solve this problem, many *gradient aggregation rules* (GARs) have been proposed to tolerate a (maximum) number  $f$  of Byzantine workers (as we discuss later in “Related work”). They can be classified in two main families: *full-GARs*, that select and average gradients of responsive workers keeping the whole information on the descent direction, and *blended-GARs*, that perform coordinate-wise operations on the set of collected gradients, inevitably losing some information (as illustrated by Figure 2 in Section 6). The former are particularly appealing in a practical setting because, even if a GAR is devised to tolerate extreme situations and provide a reasonably good accuracy despite a large number of Byzantine workers, it is important that the GAR provides very good accuracy in most frequent situations where the number ( $s$ ) of actual Byzantine workers in an execution is small ( $s \ll f$ ). In this sense, full gradients inherently enable graceful degradation.

The motivation of this work was to ask whether it is possible to derive a full gradient aggregation rule defending against 50% of Byzantine workers ( $n > 2f$ ) with a low time complexity ( $\mathcal{O}(nd)$ ), which are both optimal, but with an angular error close to that of averaging (which is not Byzantine-resilient). We answer positively by presenting AKSEL<sup>2</sup>, a

---

<sup>2</sup> Aksel (known as Kusaila in Arabic and Caecilius in Latin) was an Amazigh leader of the 7th century



new scalable median-based approach to aggregate the gradients. Essentially, Aksel is unique in the sense that it is a full-gradient GAR using indirectly the power of coordinate-wise operations to reduce the angular error.

Looking for optimal breakdown point and time complexity is self justifying. But why seek a low angular error? In fact, this is directly linked to the quality of the solution and the speed of convergence. Intuitively, a large angle makes enough room for Byzantine workers to corrupt the machine learning model. Moreover, two models with different GARs can converge to the same solution, but with different speed. We establish in Corollary 9 the link between the angle value and the convergence slowdown occasioned by the robust GAR compared to averaging.

**Related work.** Most approaches that have been proposed to improve the Byzantine resilience of gradient descent (and its variants) rely on robust statistics, whilst some use historical information to identify correct workers. KRUM [4] selects the vector with the minimum score defined as the sum of euclidean distances with its neighbors.  $m$ -KRUM [9] consists in averaging  $m$  KRUM outputs without replacement. BULYAN [12] applies a variant of the trimmed mean on a selection of vectors obtained from  $m$ -KRUM.

MEDIAN and  $b$ -TRMEAN [31] apply robust statistics on each coordinate of the  $n$  gradients. Trimmed mean ( $b$ -TRMEAN) removes the smallest and the largest  $b$  values and averages the remaining  $n - 2b$  values, whereas the median MEDIAN is a special case of Trimmed mean where  $b = \lfloor \frac{n}{2} \rfloor$ .  $b$ -PHOCAS [29] averages the  $n - b$  closest values to  $b$ -TrMean in each coordinate. MEAMED [28] is a special case of Phocas where the trimmed mean is replaced with the median. GEOMETRIC MEDIAN OF MEANS [8] computes the average of  $m$  batches of gradients, then computes the geometric median of those averages. Since no exact algorithm is available for GEOMED, the  $(1 + \epsilon)$ -approximation is used instead. DRACO [7] uses coding theory and a redundancy scheme to aggregate the gradients. BYZANTINESGD [2] and KARDAM [10] both use historical information on the gradients and construct filters that allow to distinguish bad workers from honest ones. Recent techniques from Multidimensional approximate agreement [14, 20] are also good candidates because the output of the correct workers remains inside the convex hull of the correct workers input, which is a desirable property for the problem at hand.

The median is particularly interesting for it constitutes a straightforward mechanism to deal with outliers. Yet, although the median is guaranteed to be inside the set of correct scalar values, its multidimensional variant (Coordinate-wise Median) may not lie within the convex hull of correct vectors. Second, the median heavily protects against outliers at the expense of statistical meaning. As a matter of fact, the median throws away many interesting values which makes it less efficient, as we explain later.  $b$ -TRMEAN and  $b$ -PHOCAS are very efficient when the truncation parameter  $b$  is greater than the number of Byzantine workers. However, to defend against  $s = \lceil \frac{n}{2} \rceil - 1$ , the value of  $b$  must be equal to its upper bound and the two GARs are reduced to their special cases, namely, MEDIAN and MEAMED. Otherwise, they become as vulnerable as averaging, whose deviation under attack is unbounded. One common aspect about these blended-GARs is the fact that they defend against dimensional attacks [28] but cannot reach top accuracy in honest settings with none or few Byzantine workers. The only full-GARs proposed to this day are KRUM,  $m$ -KRUM and BULYAN. These are all powerful, but they have a high time complexity (at least  $\mathcal{O}(n^2d)$ ) and their breakdown point is far from optimal. DRACO is the only aggregation rule not suffering from

---

who resisted the conquest of North Africa while making pragmatic alliances with the Byzantines.

vulnerabilities of common statistics. However, it only defends against a very limited number of Byzantine workers because of the redundancy scheme. Also, DRACO cannot be used in settings where privacy matters, because of the matrix allocation mechanism needed before the encoding phase. BYZANTINESGD and KARDAM are different from the first category of GARs because they use information on past gradients to filter the Byzantine estimates. Although theoretical guarantees have been provided for convergence, BYZANTINESGD requires too many parameters to be tuned, which make it less practical. KARDAM is the only GAR tolerating asynchrony, but it only works for Lipschitz loss functions, and defends only against  $n > 3f$ . Finally, multidimensional approximate agreement algorithms are round based, which means that, at each SGD iteration, many rounds ( $\mathcal{O}(\log \frac{\Delta}{\epsilon})$ ,  $\Delta$  being the initial diameter of the correct set of workers) need to be executed in order to agree on a gradient with an error rate  $\epsilon$ . These techniques may be advantageous in coordinator-free settings (fully decentralized learning). Table 1 compares various GARs to AKSEL according to several properties.

Basically, the full-GARs achieve top accuracy when  $s \ll f$  but are not optimal in terms of complexity and break down point. They also have a big angular error. In contrast, blended-GARs have optimal complexity and break down point with a small angular error, but do not achieve top accuracy when  $s \ll f$ . AKSEL achieves the best of both worlds.

■ **Table 1** Comparing the time complexity (TC), the breakdown point (BDP) and the expected angular error of gradient aggregation rules (GARs). Parameter  $f$  denotes the maximal number of Byzantine workers. Parameter  $m$  is specific to  $m$ -KRUM (which consists in averaging  $m$  KRUM outputs without replacement). Parameter  $b$  is specific to PHOCAS and TRMEAN and sets the level of truncation. AKSEL is the best full-GAR for all three properties.

GARs	TC	BDP	Angular error	
			$f = \mathcal{O}(1)$	$f = \mathcal{O}(n)$
AVERAGING	$\mathcal{O}(nd)$	$f = 0$	$\mathcal{O}(\sqrt{\frac{d}{n}})$	$\mathcal{O}(\sqrt{\frac{d}{n}})$
<i>Full-aggregathors</i>				
KRUM	$\mathcal{O}(n^2d)$	$n > 2f + 1$	$\mathcal{O}(\sqrt{nd})$	$\mathcal{O}(n\sqrt{d})$
$m$ -KRUM	$\mathcal{O}(n^2d)$	$n > 2f + 2$ $m < n - f - 2$	$\mathcal{O}(\sqrt{nd})$	$\mathcal{O}(n\sqrt{d})$
BULYAN	$\mathcal{O}(n^2d)$	$n > 4f + 2$	$\mathcal{O}(\sqrt{nd})$	$\mathcal{O}(n\sqrt{d})$
AKSEL	$\mathcal{O}(nd)$	$n > 2f$	$\mathcal{O}(\sqrt{d})$	$\mathcal{O}(\sqrt{d})$
<i>Blended-aggregathors</i>				
MEDIAN	$\mathcal{O}(nd)$	$n > 2f$	$\mathcal{O}(\sqrt{d})$	$\mathcal{O}(\sqrt{d})$
$(1 + \epsilon)$ -GEOMED	$\mathcal{O}(nd)$	$n > 2f$	$\mathcal{O}(\sqrt{nd})$	$\mathcal{O}(\sqrt{nd})$
b-PHOCAS	$\mathcal{O}(nd)$	$n > 2f$ $b > f$	$\mathcal{O}(\sqrt{\frac{d}{n}})$	$\mathcal{O}(\sqrt{d})$
b-TRMEAN	$\mathcal{O}(nd)$	$n > 2f$ $b > f$	$\mathcal{O}(\sqrt{\frac{d}{n}})$	$\mathcal{O}(\sqrt{d})$
MEAMED	$\mathcal{O}(nd)$	$n > 2f$	$\mathcal{O}(\sqrt{d})$	$\mathcal{O}(\sqrt{d})$

**Contributions.** We present in this paper AKSEL, a new median based algorithm which is the first to have the 4 following properties simultaneously:

- Optimal time complexity  $\mathcal{O}(nd)$
- Optimal breakdown point  $n > 2f$
- Full gradient aggregation (high accuracy reachable for  $s \ll f$ )
- Constant upper bound ( $\mathcal{O}(d)$  in the number of workers  $n$ , see Lemma 10) on the expected angular error (scalability)



On the theoretical side, we prove (1) the  $(\alpha, f)$ -Byzantine resilience of AKSEL; (2) its convergence for non convex and strongly convex losses; and (3) a logarithmic upper bound of the real angular error of AKSEL.

On the practical side, we report on an empirical evaluation of our distributed implementation of AKSEL. In particular, we consider two state-of-the-art attacks [3, 30] on academic classification tasks (MNIST, Fashion-MNIST and CIFAR-10). AKSEL reaches the top accuracy when  $s \ll f$ , and maintains a good accuracy in the extreme case  $s = f$ . AKSEL does also have some advantages that may appeal to practitioners: it requires no parameter tuning for the aggregation (a time consuming task in general) and no knowledge of the number of Byzantine workers (which can be fatal if underestimated, e.g. b-PHOCAS and b-TRMEAN). AKSEL is also based on simple mathematical functions (i.e. median, subtraction, sum-of-squares, averaging) which makes it simple to analyze.

A recent paper [11] proposed a genuinely distributed scheme with multiple servers, tolerating the Byzantine failures of a fraction of them by composing established GARs such as KRUM, m-KRUM and BULYAN. For pedagogical reasons, we present here AKSEL in a single-server setting, focusing on improving resilience to failures of workers. However, AKSEL satisfies the properties required by [11] from a GAR, and could therefore be used also in a multi-server setting instead of KRUM, m-KRUM and BULYAN in [11].

**Outline.** The paper is organized as follow. We first present our model in Section 2. After some preliminaries in Section 3, we motivate the design of our algorithm and present it in Section 4. Theoretical guarantees on its Byzantine resilience and convergence are presented in Section 5. Section 6 reports on a selection of empirical results. We conclude the paper by discussing some open issues in Section 7. For space limitations, we defer all the proofs and the full empirical evaluation to the appendix.

## 2 Model

As discussed previously, most machine learning algorithms use gradient descent (GD) to minimize a cost function  $F(\mathbf{w}_t)$  where  $\mathbf{w}_t$  is a vector of parameters<sup>3</sup> at time  $t$ . Typically, the cost function is a sum of individual errors run through many examples of the data set. Vanilla GD runs the sum through the entire dataset. However, this takes a lot of time to compute, and this is not realistic with huge datasets involving hundreds of billions of examples. Another variant is the stochastic gradient descent algorithm (SGD) which only uses a single example in each iteration. This method is very fast but noisy. A compromise is to construct a mini batch, namely a small subset of the dataset, run the sum of individual errors over this mini batch, and compute the gradient. A randomly sampled mini batch typically contains redundant examples, which can be useful to smooth out noisy gradients. Mini batch SGD is a good choice to compute quality gradients in a reasonable time. Besides, mini batch SGD is highly parallelizable. One can generate  $n$  mini batches and compute  $n$  gradients, then average them to get a very good estimate of the true gradient. In a distributed setting, randomly sampled mini batches are allocated to  $n$  workers (compute nodes), and a server aggregates those gradients then updates the parameter vector  $\mathbf{w}_t$ .

---

<sup>3</sup> For instance, the weights and biases of a neural network.

## 2.1 Distributed SGD

We follow the classical distributed SGD model [1] where a parameter server (PS) broadcasts, in each *synchronous* round  $t$ , the parameter vector  $\mathbf{w}_t \in \mathbb{R}^d$  to  $n$  workers. We consider that  $f$  among these  $n$  workers can be Byzantine. Each correct worker  $i$  computes an estimate  $\mathbf{V}_i^t = \mathbf{G}(\mathbf{w}_t, \xi_i^t)$  of the gradient  $\nabla F(\mathbf{w}_t)$  of the cost function  $F$ , where  $\xi_i^t$  is an independent and identically distributed (i.i.d.) random variable representing the subset of the dataset, drawn randomly for worker  $i$ . The PS aggregates the  $n$  received gradients  $(\mathbf{V}_1^t, \mathbf{V}_2^t, \dots, \mathbf{V}_n^t)$  using its choice function  $\mathcal{A}$  called *aggregation rule*, then updates the parameter vector using the following SGD equation:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \gamma \mathcal{A}(\mathbf{V}_1^t, \mathbf{V}_2^t, \dots, \mathbf{V}_n^t)$$

where  $\gamma$  is an arbitrary constant called *learning rate*.

## 2.2 Adversary

A Byzantine worker has the full knowledge of the system, including the aggregation rule and the vectors proposed by other workers. It can collude with other Byzantine workers to perform attacks against the aggregation rule and prevent convergence, or make the model converge to ineffective solutions. The Byzantine workers can for instance send arbitrary values, strategically-chosen values that exploit the environment, or null values corresponding to a classical crash failure. Since we are working in a synchronous system, when a vector is not received, the PS assumes that it is a null vector. Dimensional attacks were presented in [28], meaning that corruption can happen anywhere in the gradient matrix as long as each dimension contains a majority of correct values. However, we believe that such scenario could be avoided by introducing cryptography schemes (e.g. RSA signatures / AES encryption and decryption / Diffie-Hellman secure exchange of keys. . .) to make sure that impersonation is not possible, and keep the same threat model as in [4].

## 2.3 Assumptions

We now state the (rather standard) assumptions made in this paper by default: in the rest of the paper, all assumptions, except Assumption 5, are always assumed to be true, unless specified otherwise.

► **Assumption 1.** (*Breakdown point*) *The number of Byzantine workers is strictly less than the number of correct ones:  $n > 2f$*

► **Assumption 2.** (*Smoothness*)  *$F$  is  $L$ -smooth:*  
 $\forall \mathbf{w}', \mathbf{w}, \|\nabla F(\mathbf{w}') - \nabla F(\mathbf{w})\| \leq L \|\mathbf{w}' - \mathbf{w}\|$

► **Assumption 3.** (*Strong convexity*)  *$F$  is  $K$ -strongly convex:*  
 $\forall \mathbf{w}', \mathbf{w}, F(\mathbf{w}') \geq F(\mathbf{w}) + \langle \nabla F(\mathbf{w}), \mathbf{w}' - \mathbf{w} \rangle + \frac{K}{2} \|\mathbf{w}' - \mathbf{w}\|^2$

► **Assumption 4.** (*Bounded variance and unbiased estimators*) *The proposed vectors are unbiased estimates of the true gradient and their variance is bounded:*  
 $\forall i \in \{1, \dots, n\}, \mathbb{E} \mathbf{V}_i = \nabla F$  and  $\mathbb{E} \|\mathbf{V}_i - \nabla F\|^2 < d\sigma^2$

► **Assumption 5.** (*Normal distribution; not a default assumption*) *The proposed vectors are normally distributed around the true gradient  $\nabla F$ :  $\forall i \in \{1, \dots, n\}, \mathbf{V}_i \sim \mathcal{N}(\nabla F, \boldsymbol{\sigma}^2)$  where  $\boldsymbol{\sigma}^2 = \text{diag}(\sigma^2)$  is a  $d \times d$  diagonal covariance matrix.*

Assumption 1 is very common in synchronous distributed systems. It is however worth noting that beyond the classical impossibility results in distributed computing, this assumption is a direct consequence of another impossibility result in robust statistics [24], even when all the operations are done in a *single machine*. Assumptions 2 and 4 are common in the SGD literature [5] and Assumption 3 is typically needed to prove convergence rates [6]. We also analyze AKSEL and median based GARs in general under Assumption 5. This assumption is substantiated by recent empirical findings in machine learning, where many normally distributed datasets naturally yield normally distributed gradients [18]. As we detail later, our experimental findings illustrate that AKSEL performs well in commonly used datasets.

### 3 Preliminaries

We recall in this section background results on the robustness of the median and the probabilistic absolute error between the extreme value and the mean of normal samples. These will also be useful when describing the properties of our algorithm. We also recall the measure of Byzantine resilience in the context of distributed SGD.

#### 3.1 Robustness of the median

Mosteller and Tukey [21] defined two types of robustness: resistance and efficiency. The first notion conveys the fact that an infinite change caused by a small part of a group has a bounded impact on the value of the estimate. The second means that the estimate is close to the optimal estimate in a variety of situations and not only in a particular one. Many robust estimators have been proposed for scale and location. In this paper, we focus on the median, a robust estimator of the location which is the value that separates a sorted set into two equal parts. Formally: Let  $\mathbf{X} = (x_1, x_2, \dots, x_n)$  be a set of  $n$  values, then:

$$\text{med}(\mathbf{X}) = \arg \min_y \sum_{i=1}^n |x_i - y|$$

In high dimensions, we work with the coordinate-wise median, defined as follow: Let  $\mathbf{M} = (\mathbf{V}_1, \mathbf{V}_2, \dots, \mathbf{V}_n)$  be a matrix with  $n$  column vectors  $\mathbf{V}_i = (v_{1i}, v_{2i}, \dots, v_{di})^T$  in  $\mathbb{R}^d$ , then:  $\text{MEDIAN}(\mathbf{M}) = (m_1, m_2, \dots, m_d)^T$ , where  $m_j = \text{med}(v_{j1}, v_{j2}, \dots, v_{jn}), \forall j \in [1, \dots, d]$ .

The median has high efficiency for normal data (64%) [21], and most importantly, an optimal breakdown point (50%). The last point implies that corrupting 50% of the data will have only limited impact on the location parameter. Moreover, as known from the works on Byzantine tolerant approximate agreement and clock synchronization, the median always lies inside the subset of correct values when more than 50% of the data is correct. To formalize this, we restate Lemma 4 from [28] without proof.

► **Lemma 6.** *For a sequence composed of  $f$  Byzantine values and  $n - f$  correct values  $x_1, x_2, \dots, x_{n-f}$ , if  $f \leq \lfloor \frac{n}{2} \rfloor - 1$  (the correct values dominates the sequence), then the median value  $m$  of this sequence satisfies  $m \in [x_{\min}, x_{\max}]$ .*

#### 3.2 Distribution of extreme normal values

The maximum or the minimum values observed when drawing normal samples changes when  $n$  takes different values. The extreme value theory [15] shows that the extreme values of a normal distribution follows a Gumbel distribution, depending on the number  $n$  of samples drawn. Thanks to the symmetry of our problem, we only discuss the maximum

value. Formulas for the minimum are derived in a similar way. Kotz and Nadarajah [19] show that the distribution of the maximum of  $n$  samples drawn from a standard normal random variable  $\mathcal{N}(0, 1)$  with a standard normal quantile function  $\Phi^{-1}(x)$  has the following statistics. Let  $\mu_m(n), \sigma_m(n), q_m^p(n)$  be the mean, the standard deviation and the  $p^{\text{th}}$  quantile of the maximum distribution when  $n$  samples are drawn from a standard normal distribution. Then:

$$\begin{aligned}\mu_m(n) &= \Phi^{-1}\left(1 - \frac{1}{n}\right) \\ \sigma_m(n) &= \Phi^{-1}\left(1 - \frac{1}{ne}\right) - \mu_m(n) \\ q_m^p(n) &= \mu_m(n) - \sigma_m(n) \log(-\log(p))\end{aligned}\quad (1)$$

We use these results to compute a probabilistic bound of the gap between the mean and the maximum value of  $n$  normal samples.

► **Lemma 7.** *Let  $\mathbf{X} = \{\mathbf{x}_1, \dots, \mathbf{x}_{n-f}\}$  be a set of column vectors drawn from a multivariate normal random variable  $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\sigma}^2)$  with  $\boldsymbol{\mu} = (\mu_1, \dots, \mu_d)^T$  and the covariance matrix  $\boldsymbol{\sigma} = \text{diag}_{n \times d}(\sigma)$ . Let  $\mathbf{B} = \{\mathbf{b}_1, \dots, \mathbf{b}_f\}$  be a set of arbitrary column vectors and  $(n, f) \in \mathbb{N}^2$ . Let  $\mathbf{S} = \mathbf{X} \cup \mathbf{B}$  and  $\mathbf{M} = \text{Median}(\mathbf{S})$ . Let  $\mathcal{E}_k$  be the following event for the  $k^{\text{th}}$  coordinate:  $|\mathbf{M}[k] - \boldsymbol{\mu}[k]| \leq \lambda(n, p)$ . We then have,  $\forall p \in [0, 1)$  and  $\forall n \in \mathbb{N}$ :  $P\left[\bigwedge_{k=1}^d \mathcal{E}_k\right] = p$ , where*

$$\lambda(n, p) = \Phi^{-1}\left(1 - \frac{1}{n}\right) \left(1 + \log\left[-\log(p^{\frac{1}{d}})\right]\right) - \Phi^{-1}\left(1 - \frac{1}{ne}\right) \left(\log\left[-\log(p^{\frac{1}{d}})\right]\right)$$

### 3.3 Measuring the Byzantine resilience of GARs

We make use of the now classical metric to evaluate the Byzantine resilience of gradient aggregation rules [4, 12, 9, 28]. This metric encompasses two conditions. First, as long as a proposed vector lies inside a cone around the true gradient, with an angle less than  $\frac{\pi}{2}$  (first condition), and as long as its statistical moments are controlled by the moments of the (correct) gradient estimator  $\mathbf{G}$  (second condition), this vector can be considered correct and will make a step toward the minimum of the function being optimized using SGD. The second condition allows to transfer the control (classically expressed as bounds on the moments of the gradient estimator  $\mathbf{G}$  [5]) of the discrete nature of the SGD dynamics to the choice function  $\mathcal{X}$ . Below, we recall the definition of  $(\alpha, f)$ -Byzantine resilience (introduced in [4]):

► **Definition 8.** *Let  $0 < \alpha < \frac{\pi}{2}$  be any angular value and  $f \in \{0, \dots, n\}$ . Let  $\mathbf{V}_1, \dots, \mathbf{V}_n$  be any independent identically distributed random vectors in  $\mathbb{R}^d$  with  $\mathbb{E} \mathbf{V}_i = \mathbf{G}, \forall i \in \{1, \dots, n\}$ . Let  $\mathbf{B}_1, \dots, \mathbf{B}_f$  be any random vectors in  $\mathbb{R}^d$ , possibly dependent on the  $\mathbf{V}_i$ 's. A choice function  $\mathcal{X}$  is said to be  $(\alpha, f)$ -Byzantine resilient if, for any  $1 \leq j_1 < \dots < j_f \leq n$ , the vector  $\mathcal{X} = \mathcal{X}(\mathbf{V}_1, \dots, \underbrace{\mathbf{B}_{j_1}}, \dots, \underbrace{\mathbf{B}_{j_f}}, \dots, \mathbf{V}_n)$  satisfies the following two conditions:*

- **Condition (i):**  $\langle \mathbb{E} \mathcal{X}, \mathbf{G} \rangle \geq (1 - \sin \alpha) \|\mathbf{G}\|^2$
- **Condition (ii):** for  $r = 2, 3, 4$ ,  $\mathbb{E} \|\mathcal{X}\|^r$  is bounded above by a linear combination of terms of the form  $\mathbb{E} \|\mathbf{G}\|^{r_1} \dots \mathbb{E} \|\mathbf{G}\|^{r_{n-1}}$  with  $r_1 + \dots + r_{n-1} = r$

Generally, *condition (i)* can be proved by showing that  $\mathbb{E} \mathcal{X}$  belongs to the ball centered at  $\mathbf{G}$  with radius  $r = \eta(\cdot) \sqrt{d} \sigma$  (formally:  $\|\mathbb{E} \mathcal{X} - \mathbf{G}\| < \eta(\cdot) \sqrt{d} \sigma$ ), where  $\eta(\cdot)$  is a positive function,  $d$  is the dimension of the model and  $\sigma$  is the standard deviation of the gradient estimator.

► **Corollary 9.** *The function  $\eta(\cdot)$  is positively correlated to the slowdown of convergence speed occasioned by the aggregation rule  $\mathcal{X}$  compared to averaging.*

## 4 The AKSEL Algorithm

We present our aggregation protocol AKSEL in 4.1, discuss the rationale behind its design in 4.2 and give its time complexity in 4.3.

### 4.1 Aksel

■ **Algorithm 1** AKSEL: Scalable gradient aggregation rule.

---

**Input:**  $\mathbf{V} = (\mathbf{V}_1, \mathbf{V}_2, \dots, \mathbf{V}_n)$ :  $d \times n$  matrix (received gradients)  
**Output:**  $\mathbf{Y}$ :  $d \times 1$  vector

```

/* Computing the sum of squares of each column vector  $\mathbf{V}_i$  centered around the
coordinate-wise median */
1 Let  $\mathbf{S}$  be a row vector ( $1 \times n$ ) and  $\mathbf{M}$  a column vector ( $d \times 1$ )
2  $\mathbf{M} = (\mathbf{M}[1], \mathbf{M}[2], \dots, \mathbf{M}[d])^T =$  coordinate-wise median vector constructed from  $\mathbf{V}$ 
3  $\mathbf{S} = (\sum_{j=1}^d (\mathbf{V}_1[j] - \mathbf{M}[j])^2, \sum_{j=1}^d (\mathbf{V}_2[j] - \mathbf{M}[j])^2, \dots, \sum_{j=1}^d (\mathbf{V}_n[j] - \mathbf{M}[j])^2)$ 
/* Constructing a robust interval */
4 Let  $r$  be the median of the set  $\mathbf{S}$ 
5 Let  $\mathbf{I} = [0, r]$ 
/* Averaging the new subset of column vectors from  $\mathbf{V}$  */
6 Let  $\mathbf{N}$  be the subset of vectors  $\mathbf{V}_i$ 's such that  $\|\mathbf{V}_i - \mathbf{M}\|^2 \in \mathbf{I}$  and  $|\mathbf{N}| = p$ 
7  $\mathbf{Y}[j] = \frac{1}{p} \sum_{\mathbf{V}_i \in \mathbf{N}} \mathbf{V}_i[j], \quad \forall j \in \{1, \dots, d\}$ 

```

---

### 4.2 Rationale

The goal of any aggregation rule is to produce a vector as close as possible from the true gradient of the cost function. This puts conditions on the norm as well as on the direction of the aggregated vector. Clearly, any rule that focuses only on the vectors norms comparison will not succeed because of the vulnerabilities of  $l_p$ -norms, as pointed in [12]. For example,  $V_{Correct} = (2, 2, 2, 2, 5, 5, 5, 3)^T$  and  $V_{Byzantine} = (10, 0, 0, 0, 0, 0, 0, 0)^T$  are two vectors with the same norm and very different coordinates. One way to address this issue is to add a constraint on the coordinates of all vectors by centering them around a robust location estimator. We choose the coordinate-wise median in this work.

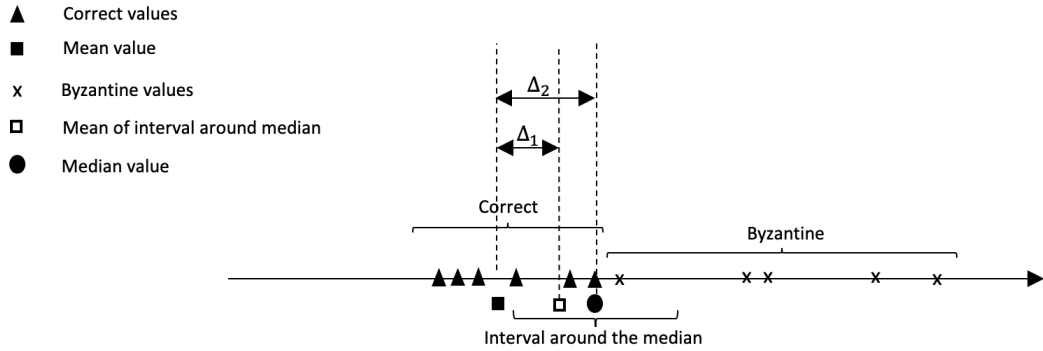
Since MEDIAN is a *blended-GAR* and provides only one aggregate which is very far from the correct mean, we choose to incorporate more vectors in the aggregation process. Therefore, a better alternative is to choose an interval around the median, and to average the values within this interval. Using this alternative, we are guaranteed to produce, most of the times, an aggregated value that lies between the real mean and the deviated median. Figure 1 illustrates the idea that an interval is better than a single value. Many GARs used this concept on each coordinate to improve the defense mechanism [31, 28, 29]. However, operating on each coordinate has consequences on the overhead cost of the Byzantine resilience. As a matter of fact, coordinate-wise operations lead to a blended vector which is different (in structure) from the full gradients. As a consequence, the top accuracy is never reached even in honest environments ( $s = 0$ ).

AKSEL is unique in the sense that it is a full-gradient GAR using indirectly the power of coordinate-wise operations. It performs the filtering method on the squared norms of the centered vectors, rather than selecting the mean around the median in each coordinate, in order to aggregate full gradients. Since norms are positive, the filter interval will be  $[0, r]$ , where  $r$  is the median of norms in our work.

The idea of centering the vectors around their coordinate-wise median is a very powerful guardrail against the vulnerability of norms, and also a very handy tool for the proof development. In fact, it is hard to come up with a probability density function for the sum of squares of normal variables with nonzero expectation, although a recent work [13] has shown that it is possible to derive a complex cumulative distribution function (but no elementary expression for the density function). Subtracting a scalar from each coordinate makes us close enough to normal variables with zero expectation, whose sum of squares density function is known and expressed through elementary expressions. We derive in Lemma 16 the expectation and the variance of the sum of squares of normal samples centered around a scalar (which, in our case, is equal to the median of the  $n$  values) for each coordinate.

### 4.3 Complexity

Our AKSEL aggregation rule has an optimal time complexity  $\mathcal{O}(nd)$ . First, AKSEL computes the coordinate-wise median ( $M$ ) in  $\mathcal{O}(nd)$  steps. Next, it subtracts  $M$  from all  $n$  gradients and computes their euclidean norms, also in  $\mathcal{O}(nd)$  steps. Then, AKSEL computes the median ( $m$ ) of the  $n$  norms using a *Quickselect* [17] in  $\mathcal{O}(n)$  steps. Finally, it averages the vectors whose norm is less than ( $m$ ) in  $\mathcal{O}(nd)$  steps. The global time complexity is therefore  $\mathcal{O}(nd)$ .



**Figure 1** Comparison of (1) the median and (2) the mean of an interval around the median, in terms of distance to the mean. In a setting where the number of Byzantine workers is exactly the number of correct workers minus one, and their values are all positioned in an extremum side, the median is always the farthest correct value from the mean among correct values. However, taking the average of values inside an interval around the median can reduce the distance to the mean value in many situations.

## 5 Theoretical Guarantees of Aksel

We give an upper bound on the variance of AKSEL and prove its  $(\alpha, f)$ -Byzantine resilience as well as its convergence properties for non convex as well as strongly convex losses.

### 5.1 Bounded variance

The following lemma states an upper bound of the variance of AKSEL.

► **Lemma 10.** *Let  $\mathbf{V}_1, \dots, \mathbf{V}_n$  be any random  $d$ -dimensional vectors in  $\mathbb{R}^d$ ,  $f$  among them being possibly Byzantine. Under Assumptions 1 and 4, the variance of AKSEL is upper bounded, and we have:*

$$\mathbb{E} \|\mathbf{A} - \nabla F\|^2 \leq \left( 4 + \frac{12 \lceil \frac{n}{2} \rceil (n - f)}{(n - \lceil \frac{n}{2} \rceil - f + 1)^2} \right) d\sigma^2 \sim \mathcal{O}(d)$$

## 5.2 Byzantine resilience

Following Definition 8, the  $(\alpha, f)$ -Byzantine resilience of AKSEL can be proved by showing first that the aggregated vector  $\mathbb{E} \mathbf{A}$  is pointing in the same direction and has a close norm to the true gradient  $\nabla F$  (*condition i*) and its statistical moments are controlled by a linear combination of the statistical moments of the correct gradient estimator (*condition ii*). We prove the two conditions through the following lemmas.

► **Lemma 11** (Expected angular error). *If Assumptions 1 and 4 hold, the angular error of AKSEL is upper bounded as follow:  $\|\mathbb{E} \mathbf{A} - \nabla F\|^2 \leq \eta^2(n, f)d\sigma^2$  where:*

$$\eta^2(n, f) = 4 + \frac{12\lceil \frac{n}{2} \rceil (n - f)}{(n - \lceil \frac{n}{2} \rceil - f + 1)^2}$$

► **Lemma 12** (Controlled statistical moments). *If Assumptions 1 and 4 hold, the statistical moments of AKSEL are upper bounded by a linear combination of the statistical moments of the correct gradient estimator:*

$$\mathbb{E} \|\mathbf{A}\|^r \leq C \sum_{r_1 + \dots + r_{n-f} = r} \|G\|^{r_1} \dots \|G\|^{r_{n-f}}$$

We now present the  $(\alpha, f)$ -Byzantine resilience result in the following theorem:

► **Theorem 13.** *Let  $\mathbf{V}_1, \dots, \mathbf{V}_n$  be a set of gradient estimates in  $\mathbb{R}^d$ . Under Assumptions 1 and 4, if  $\eta(n, f)\sqrt{d}\sigma < \|\nabla F\|$ , then AKSEL is  $(\alpha, f)$ -Byzantine resilient where  $\alpha \in [0, \frac{\pi}{2}]$  is defined by:  $\sin \alpha = \frac{\eta(n, f)\sqrt{d}\sigma}{\|\nabla F\|}$*

## 5.3 Convergence for non convex losses

When analyzing optimization algorithms under the non convexity assumption, the objective function can have several local minima instead of one global minimum. A simple solution would be to partition the parameter space into many convex pools and proceed as in the convex case. Bottou [5] proposes however to study the convergence of the objective function and its gradient instead of the parameter vector itself. When some conditions are met regarding the cost function being minimized and the learning rate, SGD converges almost surely to a flat region, where the gradient is very small. Blanchard et al. [4] combine this result with the  $(\alpha, f)$ -Byzantine resilience framework to derive a second result on the almost sure convergence of SGD using an  $(\alpha, f)$ -Byzantine resilient aggregation rule. Since AKSEL is Byzantine resilient, as proven in Theorem 13, we only restate the convergence result without proof in Theorem 14. The reader is kindly referred to [4] and [5] for more details on the convergence analysis.

► **Theorem 14.** *Let  $\mathbf{A}_t$  be the output of the AKSEL aggregation rule over the  $n$  received gradients  $\mathbf{V}_i \sim \mathbf{G}$ . We assume that (i) the cost function  $F$  is three times differentiable with continuous derivatives and is non negative ( $F(\mathbf{w}) \geq 0$ ); (ii) the learning rate satisfies  $\sum_t \gamma_t = \infty$  and  $\sum_t \gamma_t^2 < \infty$ ; (iii) the gradient estimator satisfies  $\mathbb{E} \mathbf{G}(\mathbf{w}) = \nabla F(\mathbf{w})$  and  $\forall r \in \{2, 3, 4\}, \mathbb{E} \|\mathbf{G}(\mathbf{w})\|^r \leq A_r + B_r \|\mathbf{w}\|^r$ ; (iv) there exists a constant  $0 \leq \alpha \leq \frac{\pi}{2}$  such that  $\forall \mathbf{w}, \eta(n, f)\sqrt{d}\sigma \leq \|\nabla F(\mathbf{w})\| \sin \alpha$ ; (v) finally, beyond a certain horizon  $\|\mathbf{w}\|^2 \geq D$ , there exist  $\epsilon > 0$  and  $0 \leq \beta \leq \frac{\pi}{2} - \alpha$  such that:*

$$\begin{aligned} \|\nabla F(\mathbf{w})\| &\geq \epsilon \\ \frac{\langle \mathbf{w}, \nabla F(\mathbf{w}) \rangle}{\|\mathbf{w}\| \|\nabla F(\mathbf{w})\|} &\geq \cos \beta \end{aligned}$$

*Then, the sequence of gradients  $\nabla F(\mathbf{w}_t)$  converges almost surely to zero.*



## 5.4 Convergence for strongly convex losses

Finally, we derive the statistical error rate of SGD using AKSEL as an aggregation rule.

► **Theorem 15.** *Let  $F(\mathbf{w})$  be the cost function being optimized,  $\nabla F(\mathbf{w})$  its actual gradient and  $\mathbf{A}$  the output of the AKSEL aggregation rule over the  $n$  received gradients. When Assumptions 1, 2, 3 and 4 hold, then after  $T$  iterations of SGD updates using the AKSEL GAR with a step size  $\alpha_t = \frac{1}{L}$ , we have:*

$$\mathbb{E} \|\mathbf{w}_T - \mathbf{w}_*\| \leq \left(1 - \frac{K}{L+K}\right)^T \|\mathbf{w}_0 - \mathbf{w}_*\| + \frac{2\sqrt{\Delta}}{K}$$

$$\mathbb{E}[F(\mathbf{w}_T) - F(\mathbf{w}_*)] \leq \frac{\Delta}{2L} + \left(1 - \frac{K}{L}\right)^T \left\| F(\mathbf{w}_0) - F(\mathbf{w}_*) - \frac{\Delta}{2L} \right\|$$

with:  $\Delta = \left(4 + \frac{12\lceil \frac{n}{2} \rceil (n-f)}{(n - \lceil \frac{n}{2} \rceil - f + 1)^2}\right) d\sigma^2$

## 5.5 Probabilistic upper bound on the real angular error of Aksel

In the previous section and in all the related work, results are derived in *expectation*. In fact, recent works only study the expected angular error, the variance (the expected squared absolute error) and the expected statistical error in convergence. Up to our knowledge, [31] is the only work addressing these quantities without expectation. More specifically, they study the two well known GARs MEDIAN and TRMEAN when applied with the gradient descent algorithm, assuming unbiased gradient estimates with bounded variance and skewness. They achieve an upper bound on the variance decreasing like  $\mathcal{O}(\frac{1}{\sqrt{n}})$  using normal approximations and Berry-Essen inequalities, but their breakdown point is very far from optimal:

$$\alpha + \sqrt{\frac{d \log(1 + nmLD)}{n(1 - \alpha)}} + 0.4748 \frac{S}{\sqrt{m}} \leq \frac{1}{2} - \epsilon$$

where  $\alpha$  is the ratio of Byzantine workers,  $n$  is the number of workers,  $m$  is the number of data points each worker has,  $D$  is the diameter of the parameter space,  $L$  is the Lipschitz constant,  $d$  is the dimension of the model and  $S$  is the skewness upper bound.

We study the optimal robustness ( $\alpha < \frac{1}{2}$ ) of AKSEL applied with stochastic gradient descent when gradients are normally distributed, and we show that the real angular error only has a logarithmic growth ( $\mathcal{O}(\sqrt{d} \log n)$ ) in the number of workers  $n$  under this assumption<sup>4</sup>.

## Expectation and variance of the squared norm of a centered vector

An important step in our algorithm is to sum the squares of all the coordinates centered around their median value. When Assumption 5 holds, it is possible to derive the expectation and the variance of this quantity using the asymptotic approximation of the Gamma distribution and simple bounding properties. We formalize this in the following lemma:

► **Lemma 16.** *Let  $X_i$  be a normal random variable where  $\mu_i$  is the mean,  $\sigma^2$  is the variance and  $m_i$  is a value such that  $|m_i - \mu_i| \leq \lambda\sigma$ . If  $Z_i = X_i - m_i$  is the new random variable  $X_i$  centered around  $m_i$  and  $S = \sum_{i=1}^d Z_i^2$ , then we have:*

$$\mathbb{E}[S] = (1 + \lambda^2)d\sigma^2$$

$$\text{var}[S] = 2d\sigma^4 (1 + 2\lambda^2\sigma^2)$$

<sup>4</sup> This result is interesting in its own right. Many median based GARs can benefit from this new analysis. In particular, MEDIAN which has been studied under non optimal robustness [31] and MEAMED whose expected angular error was shown to be growing as  $\mathcal{O}(\sqrt{nd})$  [28]



## Upper bound on the absolute error of Aksel

Note that the  $(\alpha, f)$ -Byzantine resilience and convergence theorems will be exactly the same in our new analysis. It suffices to derive the upper bound on the absolute error  $\|\mathbf{A} - \nabla F\|^2$  and use it in every appearance of  $\mathbb{E}\|\mathbf{A} - \nabla F\|^2$  in the previous results while dropping the expectation sign and introducing the probabilistic statement (with probability  $p$ ) before each result.

In the following lemma, we upper bound the absolute error between AKSEL’s output and the true gradient in the squared norm sense.

► **Lemma 17.** *Let  $\mathbf{V}_1, \dots, \mathbf{V}_n$  be any random  $d$ -dimensional vectors,  $f$  among them being possibly Byzantine. Let  $\lambda = \Phi^{-1}(1 - \frac{1}{n}) \left(1 + \log \left[-\log(p^{\frac{1}{d}})\right]\right) - \Phi^{-1}(1 - \frac{1}{ne}) \left(\log \left[-\log(p^{\frac{1}{d}})\right]\right)$ , where  $p \in [0, 1)$  is an arbitrary probability. When Assumptions 1 and 5 hold, the gap  $\|\mathbf{A} - \nabla F\|^2$  is upper bounded, and we have, with probability  $p$ :*

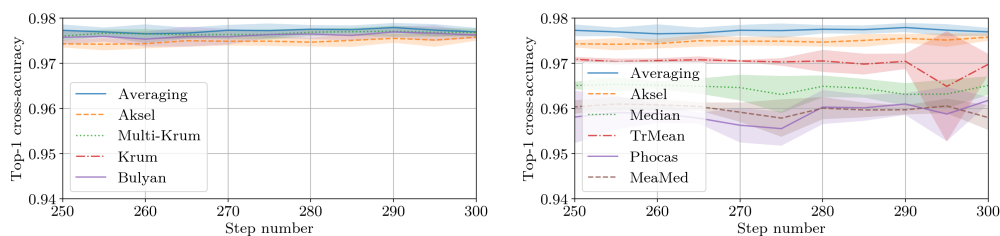
$$\|\mathbf{A} - \nabla F\|^2 \leq 2 \left[ 1 + 2\lambda^2 + \lambda \frac{\sqrt{2(1 + 2\lambda^2)}}{\sqrt{d}} \right] d\sigma^2 \sim \mathcal{O}(d \log^2 n)$$

## 6 Empirical Evaluation

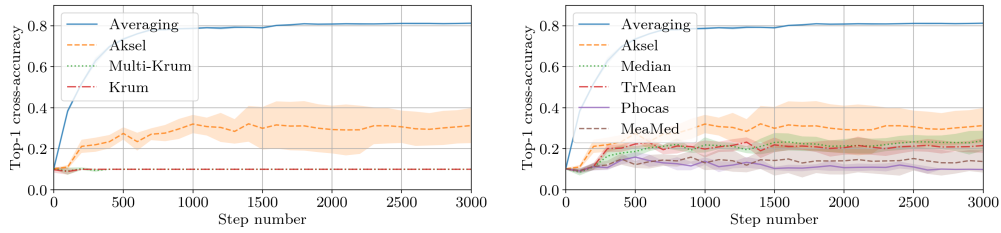
We fully implemented and evaluated AKSEL in a distributed setting. Due to space limitations, we only present here a selection of empirical results. A detailed version of the setup, as well as an extensive set of experiments, can be found in the appendix.

We tested AKSEL (and its competitors) both in settings with no Byzantine players as well as against two state-of-the-art attacks, namely “A little is enough” [3] and “Fall of empires” [30]. The first attack leverages the normal distribution of data and proposes gradients that lie within a small range containing the mean. The second attack focuses on inner product manipulation: all GARs require their inner product with the true gradient to be positive.

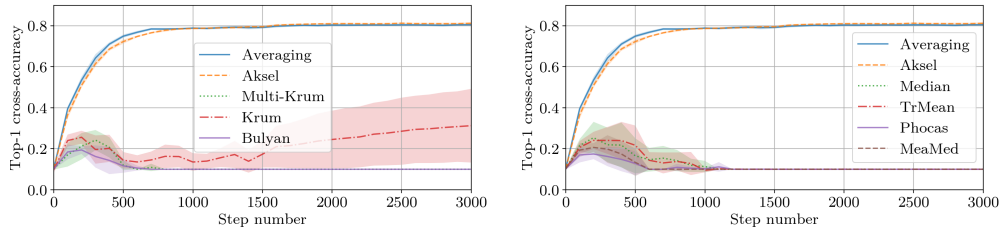
We obtained remarkable results with AKSEL, especially on complex datasets (CIFAR10). In fact, AKSEL, as any full-GAR, reaches top accuracy when  $s \ll f$  (see Figure 2). It is also able to defend against the extreme case  $s \sim f$  while maintaining a descent accuracy, thanks to its low angular error (see Figure 3). In some experiments, AKSEL is the only GAR reaching the top accuracy while others never converge (Figure 4).



■ **Figure 2** We compare AKSEL and averaging (“No Byzantine resilience”) to full-GARs (left) and blended-GARs (right) in an environment with no Byzantine worker. Here, AKSEL, as well as other full-GARs, perform as well as averaging. (MNIST dataset, using  $n = 51$  workers; the GARs are tuned to withstand up to 12 Byzantines workers.)



■ **Figure 3** CIFAR-10 using  $n = 25$  workers and  $s = f = 11$  Byzantine workers implementing attack [30]. The learning rate schedule is 0.01 for the first 1500 training steps, then 0.001 for the remaining of the training.



■ **Figure 4** CIFAR-10 using  $n = 25$  workers, including  $s = f = 5$  Byzantine workers implementing attack [3]. The learning rate schedule is 0.01 for the first 1500 training steps, then 0.001 for the remaining of the training. AKSEL is the only GAR which actually converges.

## 7 Concluding Remarks

**Summary.** This paper investigates the parameter server architecture of machine learning algorithms when trained in untrusted environments. We address time complexity, breakdown point, angular error and the overhead cost of Byzantine resilience. We propose AKSEL, the first full gradient aggregation rule with optimal time complexity and optimal breakdown point with a constant expected angular error in the number of workers. Our empirical evaluation shows that AKSEL achieves top accuracy in frequent situations with none or few Byzantine workers, while maintaining a good defense in the very few cases where the ratio of Byzantine workers approaches 50%. We also provide a new upper bound on the angular error of median based GARs (AKSEL included) which grows only in  $\mathcal{O}(\sqrt{\frac{d}{n}})$  under optimal robustness.

**Discussion.** One could also ask whether it is possible to reduce the angular error of AKSEL further and obtain that of averaging ( $\mathcal{O}(\sqrt{\frac{d}{n}})$ ), which is not Byzantine resilient. We foresee two ways to improve the angular error: either by reducing the breakdown point, which would result in a interval around the median containing only correct workers (this is the main idea of b-TRMEAN and b-PHOCAS [29]), or by sacrificing the time complexity by computing the distance between the median and the closest possible Byzantine value, which should give an idea on how tight the filtering interval should be to average only correct workers. Note that if we replace MEDIAN in AKSEL with b-TRMEAN, it is possible to reduce the expected angular error to  $\mathcal{O}(\sqrt{\frac{d}{n}})$  when  $f = \mathcal{O}(1)$ . However, we prefer the current version of AKSEL because it does not need the truncation parameter  $b$  which, if underestimated, can cause a serious problem in the training.

We see many ways to relax some of the assumptions we make in this paper. We believe for instance that the Byzantine resilience and the convergence analysis could be done using biased estimates, as in [6]. One could also derive an upper bound of the variance of gradients

using the smoothness assumption, as discussed in [22], without assuming a constant upper bound  $\sigma^2$  (as assumed in all previous papers). Another interesting direction is to leverage randomness to improve Byzantine resilience.

---

## References

---

- 1 Martin Abadi et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.
- 2 Dan Alistarh, Zeyuan Allen-Zhu, and Jerry Li. Byzantine stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 4613–4623, 2018.
- 3 Moran Baruch, Gilad Baruch, and Yoav Goldberg. A little is enough: Circumventing defenses for distributed learning, 2019. [arXiv:1902.06156](https://arxiv.org/abs/1902.06156).
- 4 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.
- 5 Léon Bottou. *On-Line Learning and Stochastic Approximations*, page 9–42. Cambridge University Press, USA, 1999.
- 6 Léon Bottou, Frank E. Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *SIAM Review*, 60(2):223–311, 2018.
- 7 Lingjiao Chen, Hongyi Wang, Zachary Charles, and Dimitris Papailiopoulos. DRACO: Byzantine-resilient distributed training via redundant gradients. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 903–912. PMLR, 2018.
- 8 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.
- 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, Richeek 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 Yuri Fateev, Vladimir Shaydurov, Evgeny Garin, Dmitry Dmitriev, and Valeriy Tyapkin. Probability distribution functions of the sum of squares of random variables in the non-zero mathematical expectations. *Journal of Siberian Federal University. Mathematics & Physics*, 9:173–179, 2016.
- 14 Matthias Függer and Thomas Nowak. Fast Multidimensional Asymptotic and Approximate Consensus. In *32nd International Symposium on Distributed Computing (DISC 2018)*, volume 121 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:16. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018.
- 15 E.J. Gumbel. *Statistics of Extremes*. Dover books on mathematics. Dover Publications, 2004.
- 16 Robert Hecht-Nielsen. Theory of the backpropagation neural network. In *Neural networks for perception*, pages 65–93. Elsevier, 1992.
- 17 C. A. R. Hoare. Algorithm 65: Find. *Commun. ACM*, 4(7):321–322, 1961.

- 18 Arthur Jacot, Franck Gabriel, and Clément Hongler. Neural tangent kernel: Convergence and generalization in neural networks. In *Advances in neural information processing systems*, pages 8571–8580, 2018.
- 19 S. Kotz and S. Nadarajah. *Extreme Value Distributions*. World Scientific Publishing Company, 2000.
- 20 Hammurabi Mendes, Maurice Herlihy, Nitin Vaidya, and Vijay Garg. Multidimensional agreement in byzantine systems. *Distributed Computing*, 28:1–19, 2015.
- 21 F. Mosteller and J.W. Tukey. *Data Analysis and Regression: A Second Course in Statistics*. Addison-Wesley Series in Behavioral Science, 1977.
- 22 Lam M. Nguyen, Phuong Ha Nguyen, Marten van Dijk, Peter Richtárik, Katya Scheinberg, and Martin Takáč. Sgd and hogwild! convergence without the bounded gradients assumption, 2018.
- 23 B. T. Polyak and A. B. Juditsky. Acceleration of stochastic approximation by averaging. *SIAM Journal on Control and Optimization*, 30(4):838–855, 1992.
- 24 Peter J Rousseeuw. Multivariate estimation with high breakdown point. *Mathematical statistics and applications*, 8:283–297, 1985.
- 25 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.
- 26 Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.
- 27 V. Vapnik. Principles of risk minimization for learning theory. In J. E. Moody, S. J. Hanson, and R. P. Lippmann, editors, *Advances in Neural Information Processing Systems 4*, pages 831–838. Morgan-Kaufmann, 1992.
- 28 Cong Xie, Oluwasanmi Koyejo, and Indranil Gupta. Generalized byzantine-tolerant sgd, 2018.
- 29 Cong Xie, Oluwasanmi Koyejo, and Indranil Gupta. Phocas: dimensional byzantine-resilient stochastic gradient descent, 2018.
- 30 Cong Xie, Oluwasanmi Koyejo, and Indranil Gupta. Fall of empires: Breaking byzantine-tolerant sgd by inner product manipulation. In *UAI*, volume 115 of *Proceedings of Machine Learning Research*, pages 261–270. PMLR, 2020.
- 31 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.

# ACE: Abstract Consensus Encapsulation for Liveness Boosting of State Machine Replication

Alexander Spiegelman

Novi Research, Menlo Park, USA

Arik Rinberg<sup>1</sup>

Technion, Haifa, Israel

Dahlia Malkhi

Novi Research, Menlo Park, USA

---

## Abstract

With the emergence of attack-prone cross-organization systems, providing asynchronous state machine replication (SMR) solutions is no longer a theoretical concern. This paper presents *ACE*, a framework for the design of such fault tolerant systems. Leveraging a known paradigm for randomized consensus solutions, *ACE* wraps existing practical solutions and real-life systems, boosting their liveness under adversarial conditions and, at the same time, promoting load balancing and fairness. Boosting is achieved without modifying the overall design or the engineering of these solutions.

*ACE* is aimed at boosting the prevailing approach for practical fault tolerance. This approach, often named *partial synchrony*, is based on a leader-based paradigm: a good leader makes progress and a bad leader does no harm. The partial synchrony approach focuses on safety and forgoes liveness under targeted and dynamic attacks. Specifically, an attacker might block specific leaders, e.g., through a denial of service, to prevent progress. *ACE* provides boosting by running *waves* of parallel leaders and selecting a *winning* leader only retroactively, achieving boosting at a linear communication cost increase.

*ACE* is agnostic to the fault model, inheriting its failure model from the wrapped solution assumptions. As our evaluation shows, an asynchronous Byzantine fault tolerance (BFT) replication system built with *ACE* around an existing partially synchronous BFT protocol demonstrates reasonable slow-down compared with the base BFT protocol during faultless synchronous scenarios, yet exhibits significant speedup while the system is under attack.

**2012 ACM Subject Classification** Security and privacy → Distributed systems security; Software and its engineering → Abstraction, modeling and modularity

**Keywords and phrases** Framework, Asynchronous, Consensus boosting, State Machine Replication

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2020.9

**Related Version** A full version of the paper is available at [49], <https://arxiv.org/abs/1911.10486>.

## 1 Introduction

Building reliable systems via state machine replication (SMR) requires resilience against all network conditions, including malicious attacks. The best way to model such settings is by assuming asynchronous communication links. However, as shown in the FLP result [27], deterministic asynchronous SMR solutions are impossible.

Two principal approaches are used to circumvent this result. The first is by assuming *partial synchrony* [25], in which protocols are designed to guarantee safety under worst case network conditions, but are able to satisfy progress only during “long enough” periods of network synchrony. Protocols in this model normally follow the leader-based view-by-view

---

<sup>1</sup> Part of this work has been done while Arik Rinberg was an intern at VMware Research Group.



paradigm due to its speed during synchronous attack-free periods and relative simplicity. In fact, most deployed systems, several of which have become the de facto standards for building reliable systems (e.g., Paxos [35], PBFT [19], Zyzzyva [34], Zookeeper [2] Raft [45] and others [45, 4, 51, 13]), adopt this approach. The drawback of the partial synchrony model is that it fails to capture adaptive network attacks [48], leaving the leader-based view-by-view algorithms vulnerable. For example, an attacker can prevent progress by adaptively blocking the communication of the leader of every view.

The second approach to circumventing the FLP impossibility is by employing randomization [12, 47, 21]. Randomized algorithms typically satisfy safety properties, but ensure liveness only with probability approaching 1, albeit operate at network speed under all network conditions. There are many theoretical works on asynchronous consensus (also called agreement) in the literature, but since they are typically very complex or inefficient, only a few of them were used in academic asynchronous SMR systems [5, 24] and we are not aware of any deployed in practice.

**Main contribution.** This paper presents *ACE*, a framework for *asynchronous boosting* that converts consensus algorithms designed according to the *leader-based view-by-view paradigm* in the partial synchrony model into randomized fully asynchronous SMR solutions. *ACE* provides boosting by running *waves* of parallel leaders and selecting a *winning* leader only retroactively. As a result, with *ACE*, a system designer can benefit twofold: (1) from the experience gained in decades of leader-based view-by-view algorithm design and system engineering, and (2) from a robust asynchronous solution that is live under attacks.

An additional feature of *ACE* is the following notion of *fairness*. Due to the unpredictability of the election mechanism, *ACE* guarantees that for each slot the probability of parties to agree on a value proposed by an honest party is at least  $1/2$ . Another important feature of *ACE* is that it is model agnostic and can be applied to any leader-based protocol in the Byzantine or crash failure model. As a result, when instantiated with a BFT protocol such as PBFT [19] we get asynchronous byzantine state machine replication, and when instantiated with a crash-failure solution like Paxos [35] or Raft [45] we get the first asynchronous SMR system tolerating any minority of failures.

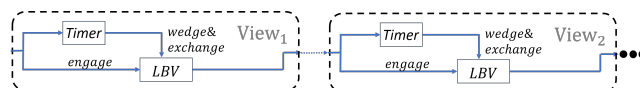
## 1.1 Technical Approach

**View-by-view paradigm.** Leader-based view-by-view protocols divide executions into a sequence of views, each with a designated leader. Every view is then further divided into two phases. First, the *leader-based* phase in which the designated leader tries to drive progress by getting all parties to commit its value, and all other parties start a timer to monitor the progress. If the timer expires before a decision is reached (due to a faulty leader or long network delays), the parties switch to the *view-change* phase. In this phase the parties *exchange* information to safely *wedge* the current view and proceed to the next one, restarting the process with the new designated leader.

**Wave-by-Wave.** Our solution boosts asynchronous liveness by eliminating the need in internal timers. To thus end, *ACE* leverages a known theoretical paradigm [31, 9] of running  $n$  leaders in parallel and retrospectively choosing one. In particular, *ACE* provides boosting by running a *wave* of  $n$  leader-base phases in parallel, waiting for a quorum of leaders to progress, and then randomly electing one leader to proceed to its *view-change* phase. Importantly, no timers are used. Rather, the trigger to move to the *view-change* phase is external and happens upon the completion of “enough” leaders.

To be able to externally switch between the phases, ACE provides a formal characterization of the leader-based view-by-view protocols by defining a *leader-based view (LBV)* abstraction, which encapsulates the main properties of a single view. Its API de-couples the *leader-based* phase from the *view-change* phase and allows each of them to be separately invoked – *engage* triggers the leader-based phase, and *wedge&exchange* triggers the view-change phase.

We define the properties of LBV, and conjecture that existing view-by-view algorithms implicitly satisfy them. Moreover, decomposing such algorithms according to the LBV abstraction yields a sequence of LBV’s with an external timer triggering phase transitions, as depicted in Figure 1.



■ **Figure 1** Using a sequence of LBV instances to reconstruct a partially synchronous leader-based view-by-view protocol.

More specifically, in a single-shot agreement protocol, an ACE wave operates as follows: Instead of running one LBV instance (as view-by-view protocols do), a wave runs  $n$  LBV instances (the leader-based phase) simultaneously, each with a distinct leader. Then, the wave performs a barrier synchronization in which parties wait until a quorum of the instances have completed. The barrier is eventually reached due to a key property of the LBV abstraction, which guarantees that if the leader is correct and no correct party invokes *view-change*, then all correct parties eventually commit a value.

After the barrier is reached, one LBV instance is selected unpredictably and uniformly at random. The chosen instance “wins”, and all other instances are ignored. Then, parties use the LBV’s *wedge&exchange* API to invoke the view-change phase in the chosen instance (only). The view-change phase here has two purposes. First, it *boosts termination*. If the chosen LBV instance has reached a decision, meaning that a significantly large quorum of parties have decided in its leader-based phase, then all correct parties learn this decision during the view-change phase. Second, as in every view-by-view protocol, the view-change phase *ensures safety* by forcing the leaders of the next wave to propose safe values.

The next wave enacts  $n$  new LBV instances, each with a different leader that proposes a value according to the state returned from the view-change phase of the chosen instance of the previous wave. Note that since parties wait for a large quorum of LBV instances to reach a decision in each wave before randomly choosing one, the chosen LBV has a constant probability of having a decision, hence, together with the termination boosting provided by the view-change phase, we get progress, in expectation, in a constant number of waves.

As to SMR, ACE implements a variant in which parties do not proceed to the next slot before they learn the decision value of the current one, but once they move to the next one they stop participating in the current slot and garbage collect all the associated resources. Deferring next slots until the current decision is known is essential for systems in which the validity of a value for a certain slot depends on all previous decision values (e.g., Blockchains). ACE’s SMR solution uses an instance of the single-shot protocol for every slot together with a forwarding mechanism to help slow parties catch-up.

**Applicability.** ACE can take any view-by-view consensus protocol designed for the partially synchronous model and transform it into an asynchronous SMR solution. In order to instantiate ACE with a specific algorithm, e.g., PBFT [19] or Paxos [35], one only needs to take a single view of the algorithm’s logic and wrap it with the LBV API. Therefore,



instantiating ACE does not require new logic implementation beyond the engineering effort of providing the API. Furthermore, ACE’s modularity provides a clean separation of concerns between safety (provided by the LBV properties) and asynchronous liveness (provided by the framework).

**Evaluation.** To demonstrate ACE, we choose to focus on the byzantine model as this is the model considered by Blockchain systems and we believe that, due to their high stakes and public infrastructures, Blockchain systems will benefit the most from a generic asynchronous SMR solution that can tolerate network attacks. We implement ACE’s algorithms in C++ and instantiate the LBV abstraction with a variant of HotStuff [51] – a state of the art BFT solution, which is currently being implemented in several commercial Blockchain systems [6]. To compare the ACE instantiation to the base (raw) HotStuff implementation, we emulate different adversarial scenarios and generate networks attacks. Our evaluation shows that while base HotStuff outperforms ACE (instantiated with HotStuff) in the synchronous failure-free case, ACE has absolute superiority during asynchronous periods and network attacks. For example, we show that byzantine parties can hinder progress in base HotStuff by targeting leaders with a DDoS attack, whereas ACE manages to commit values at network speed.

**Roadmap.** The rest of the paper is organized as follows: Section 2 describes the model and formalizes the agreement and SMR problems. Section 3 gives an overview of the leader-based view-by-view paradigm, capturing its main properties and vulnerabilities. Section 4 defines ACE’s abstractions, and its algorithms are given in Section 5. Section 6 instantiates ACE and evaluates its performance. Finally, Section 7 discusses related work and Section 8 concludes.

## 2 Model and Problem Definitions

### 2.1 System Model

We consider a peer to peer system with  $n$  parties,  $f < n$  of which may fail. We say that a party is *faulty* if it fails at any time during an execution of a protocol. Otherwise, we say it is *correct*. In a peer to peer system every pair of parties is connected with a *communication link*. A message sent on a link between two correct parties is guaranteed to be delivered, whereas a message to or from a faulty party might be lost. A link between two correct parties is *asynchronous* if the delivery of a message may take arbitrary long time, whereas a link between two correct parties is *synchronous* if there is a bound  $\Delta$  for message deliveries. In *asynchronous network periods* all links among correct parties are asynchronous, whereas during *synchronous network periods* all such links are synchronous.

A standard communication model assumed by algorithms that follow the view-by-view paradigm is the *partially synchronous* model (also called eventual synchrony [25]). In this model, there is an unknown point in every execution, called *global stabilization time (GST)*, which divides the execution into two network periods: before GST the network is asynchronous and after GST the network is synchronous. The partially synchronous model was defined to capture spontaneous network disconnections in wide-area networks, in which case it is reasonable to assume that asynchronous periods are short and synchronous periods are long enough for the protocols to make progress.

However, the partially synchronous model fails to capture malicious attacks that intentionally try to sabotage progress, and thus are not suitable for many current use cases (e.g., Blockchains). For example, one possible attack is the *weakly adaptive asynchronous* in which an attacker adaptively blocks one party at a time from sending or receiving messages (e.g.,



via DDOS). This results in a *mobile* asynchrony that moves from party to party, violating the GST assumption made by the partially synchronous model, and thus prevents progress from all leader-based view-by-view algorithms.

ACE, in contrast, assumes the fully asynchronous communication model, and thus progress in network speed under all network conditions and attacks as long as messages among correct parties are eventually delivered.

As mentioned in the Introduction and explained in more detail below, ACE abstracts away specific model assumptions and implementation details into three primitives: *Leader based view (LBV)*, *leader-election*, and *barrier*. In Section 4, we define the properties of these primitives and require that any leader-based view-by-view protocol that is instantiated into our framework satisfies them. To satisfy these properties, each protocol may have different model assumptions: for example, the relation between  $f$  and  $n$ , the failure types (e.g., crash and byzantine), and cryptographic assumptions. ACE inherits the specific assumptions made by each of the protocols it is instantiated with, and adds nothing to them. In other words, whatever assumptions are made by the instantiated protocol in order to satisfy the abstractions' properties, are exactly the assumptions under which ACE operates.

## 2.2 Problem Definition

We define the fair validated single-shot agreement problem below, and for space limitation defer the definition of the generalized SMR problem to the full paper [49].

The *fair validated agreement* [9, 16, 15] is a single-shot problem in which correct parties propose externally valid values and agree on one unique such value. The formal properties are given below:

- Agreement: All correct parties that decide, decide on the same value.
- Termination: If all correct parties propose valid values, then all correct parties decide with probability 1.
- Validity: If a correct party decides on a value  $v$ , then  $v$  is externally valid.

Note that the agreement and termination properties are not enough by to guarantee real progress of any multi-shot agreement system (e.g., Blockchain) that is built on top of the single-shot problem. Without external validity, parties are allowed to agree on some pre-defined value (i.e.,  $\perp$ ) [43], which is basically an agreement not to agree. Moreover, as long as a value satisfies the system's external validity condition (e.g., no contradicting transactions in a blockchain system), parties may decide on this value even if it was proposed by a byzantine party. However, since high stake is involved and byzantine parties may try to increase the ratio of decision values proposed by them, we require an additional fairness property that is a generalization of the *quality* property defined in [9]:

- Fairness: The probability for a correct party to decide on a value proposed by a correct party is at least  $1/2$ . Moreover, during synchronous periods, all correct parties have an equal probability of  $1/n$  for their values to be chosen.

Intuitively, note that by simply following the protocol byzantine parties can have a probability of  $1/3$  (recall that  $1/3$  of the parties are byzantine) for their value to be chosen in every protocols even during synchronous periods. And since during asynchronous periods the adversary can, in addition, block  $1/3$  of the correct parties, we get that byzantine parties can increase their probability to  $1/2$ . Meaning that the fairness property we require is optimal.

### 3 The View-by-View Paradigm

Many (if not all) practical agreement and consensus algorithms operate a leader-based view-by-view paradigm, designed for partially synchronous models, including the seminal work of Dwork et al. [25] pioneering the approach, and underlying classical algorithms like Paxos [35], Viewstamped-Replication [44], PBFT [19], and others [34, 45].

Protocols designed according to the view-by-view paradigm advance in views. Each view has a designated leader that proposes a value and tries to convince other parties to decide on it. To tolerate faulty leaders from halting progress forever, parties use timers to measure leader progress; if no progress is made they demote the leader, abandoning the current view and proceeding to the next one.

The main problem with this approach is that a faulty leader that does not send any messages is indistinguishable from a correct leader with asynchronous links. Therefore, protocols implementing this approach are not able to guarantee progress during asynchronous periods or weakly adaptive asynchronous attacks since parties advance views before correct leaders are able to drive decisions. Below we discuss the main properties of algorithms designed according to the view-by-view paradigm:

#### 3.1 Main properties

**Safety.** Perhaps the most important property of such algorithms is their ability to satisfy safety during arbitrary long asynchronous periods. This is achieved via a careful *view-change* mechanism that governs the transition between views. View-change consists of parties *wedging* the current view by abandoning the current leader, and *exchanging* information about what might have committed in the view (the closing state of the view). In the new view, parties participate in the new leader's phase only if it proposes a value that is safe in accordance with the closing state.

**Liveness.** Algorithms that rely on leaders to drive progress cannot guarantee progress during asynchronous periods since they cannot distinguish between faulty leaders and correct ones with asynchronous links. During asynchronous periods, messages from the current leader may be delivered only after parties timeout and move to the next view regardless of how conservative the timeouts are.

However, all these algorithms share an important property that our framework utilizes: for every view, if the leader of the view is correct and no correct party times out and abandons this view, then all correct parties decide in this view.

#### 3.2 Practical Vulnerabilities

Deploying view-by-view algorithms requires tuning the leader timeouts. On the one hand, aggressive timeouts set close to the common network delay might cause correct leaders to be demoted due to spurious delays, and destabilize the system. On the other, conservative timeouts implies delayed actions in case of faulty leaders. It further opens the system to possible attacks by byzantine leaders that slow system progress to the maximum possible without triggering a timeout.

Another attack on the progress of leader-based protocols is the weak adaptive asynchrony in which an attacker blocks communication with the leader of each view until the view expires, e.g., via distributed denial-of-service attack. Last, a carefully executed adaptive asynchrony attack can cause a fairness bias. Some leaders (possibly byzantine) may be

allowed to progress and commit their values, whereas an attacker blocks communication with other designated (possibly all correct) leaders. In Section 6, we demonstrate the above attacks, and show that ACE is resilient against them.

## 4 Framework abstractions

ACE provides “asynchronous boosting” for partially synchronous protocols designed according to the leader-based view-by-view paradigm. In a nutshell, ACE takes such a protocol, encapsulates a single view of the protocol into a *leader-based view (LBV)* abstraction that provides API to avoid timeouts, composes LBVs into a wave of  $n$  instances running in parallel, interjects auxiliary actions in between successive waves, and chooses one LBV instance retrospectively at random. Detailed description is given in the next section. Section 4.1 defines the *Leader based view (LBV)* abstraction and the auxiliary abstractions utilized by ACE, Barrier and Leader-election, are given in the full paper [49].

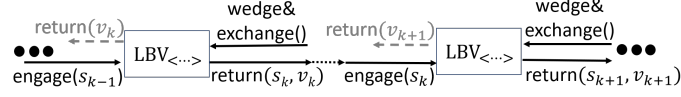
### 4.1 Encapsulating view-based agreement protocols

As explained above, each view in a leader-based view-by-view algorithm consists of two phases: First, all parties wait for the leader to perform the *leader-based* phase to drive decision on some value  $v$ , and then, if the leader fails to do it fast enough, parties switch to the *view-change* phase in which they *wedge* the current leader and *exchange* information in order to get the closing state of the view. To decide when to switch between the phases, existing algorithms use timeouts, which prevent them from guaranteeing progress during asynchronous periods. Therefore, in order to boost asynchronous liveness, ACE replaces the timeout mechanism with a different strategy to switch between the phases. To this end, the LBV abstraction exposes an API with two methods, `engage` and `wedge&exchange`, where `engage` starts the first phase of the view (leader-based), and `wedge&exchange` switches to the second (view-change). By exposing API with these two methods, we remove the responsibility of deciding when to switch between the phases from the view (e.g., no more timeouts inside a view) and give it to the framework, while still preserving all safety guarantees provided by each view in a leader-based view-by-view protocol.

Every instance of the LBV abstraction is parametrized with the leader’s name and with an identification  $id$ , which contains information used by the high-level agreement algorithm built (by the framework) on top of a composition of LBV instances. The `wedge&exchange` method gets no parameters and returns a tuple  $\langle s, v \rangle$ , where  $v$  is either a value or  $\perp$ ; and  $s$  is the closing state of the instance, consisting of all necessary information required by the specific implementation of the abstraction (e.g., a safe value for a leader to propose). The `engage` method gets the “closing state”  $s$  that was returned from `wedge&exchange` in the preceding LBV instance (or the initial state in case this is the first one), and outputs a value  $v$ . Intuitively, the returned value from both methods is the “decision” that was made in the LBV instance, but as we explain below, the high-level agreement algorithm might choose to ignore this value.

The safety of view-by-view algorithms strongly relies on the fact that correct parties start a new view with the closing state of the previous one. Otherwise, they cannot guarantee that correct parties that decide in different views decide on the same value. Therefore, when we encapsulate a single view in our LBV abstraction and define its properties, we consider only executions in which the LBV instances are composed one after another. Formally, we say that the LBV abstractions are *properly composed by a party  $p_i$*  in an execution if  $p_i$  invokes the `engage` of the first instance with some fixed initial state (which depends on the

instantiated protocol), and for every instance  $k > 1$ ,  $p_i$  invokes its `engage` with the state output of `wedge&exchange` of instance  $k - 1$ . In addition, we say that the LBV abstractions are *properly composed* in an execution if they are properly composed by all correct parties. Figure 2 illustrates LBV’s API and its properly composed execution.



■ **Figure 2** A properly composed execution: The `engage` method of instance  $k > 1$  gets the state output of the `wedge&exchange` method of instance  $k - 1$ .

The formal definition of the LBV abstraction is as follows:

► **Definition 1.** *A protocol implements an LBV abstraction if the following properties are satisfied in every properly composed execution that consists of a sequence of LBV instances:*

**Liveness.**

- **Engage-Termination:** *For every instance with a correct leader, if all correct parties invoke `engage` and no correct party invokes `wedge&exchange`, then `engage` invocations by all correct parties eventually return.*
- **Wedge&Exchange-Termination:** *For every instance, if all correct parties invoke `wedge&exchange` then all `wedge&exchange` by correct parties eventually return.*

**Safety.**

- **Validity:** *For every instance, if an `engage` or `wedge&exchange` invocation by a correct party returns a value  $v$ , then  $v$  is externally valid.*
- **Completeness:** *For every instance, if  $f + 1$  `engage` invocations by correct parties return, then no `wedge&exchange` invocation by a correct party returns a value  $v = \perp$ .*
- **Agreement:** *If an `engage` or `wedge&exchange` invoked in some instance by a correct party returns a value  $v \neq \perp$  and some other `engage` or `wedge&exchange` invoked in some instance by a correct party returns  $v' \neq \perp$  then  $v = v'$ .*

Note that during the view-change phase in most leader-based protocols, parties send the closing state only to the leader of the next view. However, in ACE, since we run  $n$  concurrent LBV instances, each with a different leader, we need all parties to learn the closing state after `wedge&exchange` returns. Moreover, as mentioned above and captured by the Completeness property, we use `wedge&exchange` to also boost decisions in order to guarantee that if the retrospectively chosen LBV instance successfully completed the first (leader-based) phase, than all correct parties decide at the end of its second phase. Therefore, when encapsulating the view-change mechanism of a leader-based protocol into the `wedge&exchange` method, a small change has to be made in order to satisfy the above properties. Instead of sending the closing state only to the next leader, parties need to exchange information by sending the closing state to all parties and wait to receive  $n - f$  such messages. No change is needed to the first phase of the encapsulated leader-based protocol since all the required properties for `engage` are implicitly satisfied.

**5 Framework algorithms**

In this section we present ACE’s asynchronous boosting algorithms, which are built on top of the abstractions defined above. The algorithm for an asynchronous single-shot agreement is given below, and for space limitation, we show how to turn it into an asynchronous SMR

in the full paper [49]. For completeness, the full paper [49], we show how to use the LBV abstraction to reconstruct the base partially synchronous view-by-view algorithm the LBV is instantiated with.

The pseudocode for the asynchronous single-shot agreement protocol appears in Algorithm 1 and a formal correctness proof can be found in the full paper [49]. An invocation of the protocol ( $SS\text{-propose}(id, S)$ ) gets an initial state  $S$  and identification  $id$ , where the initial state  $S$  contains all the initial specific information (including the proposed value) required by the leader-based view-by-view protocol instantiated in the LBV abstraction.

■ **Algorithm 1** Asynchronous single-shot agreement.

---

```

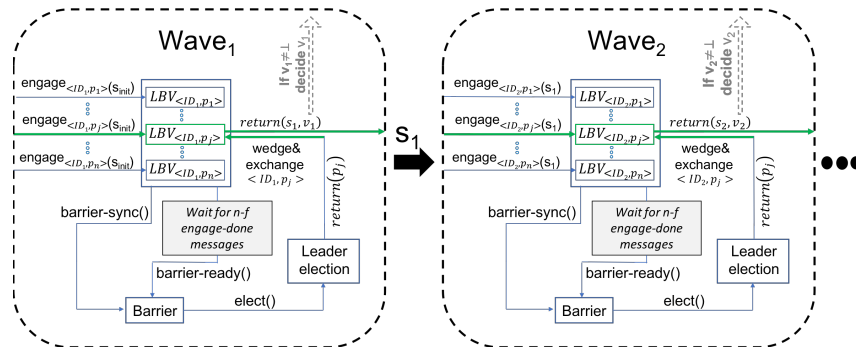
1: upon  $SS\text{-propose}(id, S)$  do
2:    $state \leftarrow S$ ;  $wave \leftarrow 1$ 
3:   while true do
4:      $ID \leftarrow (id, wave)$ 
5:      $\langle state', value \rangle \leftarrow \text{WAVE}(ID, state)$ 
6:     if  $value \neq \perp$  and did not decide before
       then
7:       decide  $(id, value)$ 
8:        $state \leftarrow state'$ 
9:        $wave \leftarrow wave + 1$ 
10:  upon  $\text{engage } \langle ID, p_j \rangle$  returns  $v$  do
11:    send "ID, ENGAGE-DONE" to party  $p_j$ 
12:  procedure  $\text{WAVE}(ID, state)$ 
13:    for all  $p_j = p_1, \dots, p_n$  do
14:      invoke  $\text{engage } \langle ID, p_j \rangle(state)$ 
        //non-blocking
15:    barrier-sync  $ID()$ 
16:     $leader \leftarrow \text{elect } ID()$ 
17:    return  $\text{wedge\&exchange } \langle ID, leader \rangle()$ 
18:  upon receiving  $n - f$  "ID, ENGAGE-DONE" do
19:    invoke  $\text{barrier-ready } ID()$ 

```

---

The protocol proceeds in a *wave-by-wave* manner. The state is updated at the end of every wave and a decision is made the first time a wave returns a non-empty value. In every wave, each party first invokes the `engage` operation in  $n$  LBV instances, each with a different leader. Each invocation gets the state obtained at the end of the previous wave or the initial state if this is the first wave.

Then, parties invoke `barrier-sync` and wait for it to return. Recall that by the B-Coordination property, `barrier-sync` returns only after  $f + 1$  correct parties invoke `barrier-ready`. When an `engage` invocation in an LBV instance with leader  $p_j$  returns, a correct party sends an "ENGAGE-DONE" message to party  $p_j$ , and whenever a party gets  $n - f$  such messages it invokes `barrier-ready`. Denote an LBV instance as *successfully completed* when  $f + 1$  correct parties completed the first phase, i.e., their `engage` returned, and note, therefore, that a correct party invokes `barrier-ready` only after the LBV instance in which it acts as the leader was successfully completed. Thus, a `barrier-sync` invocation by a correct party returns only after  $f + 1$  LBV instances successfully completed.



■ **Figure 3** Asynchronous single-shot algorithm. The chosen LBVs, marked in green, are properly composed.

Next, when the barrier-sync returns, parties elect a unique leader via the leader-election abstraction, and further consider only its LBV instance. Note that since parties wait until  $f + 1$  LBV instances have successfully completed before electing the leader, with a constant probability of  $\frac{f+1}{n}$  the parties elect a successfully completed instance (can be improved to  $\frac{2f+1}{n}$  in the byzantine case with  $n = 3f + 1$ ), and even an adaptive adversary has no power to prevent it.

Finally, all parties invoke `wedge&exchange` in the elected LBV instance to wedge and find out what happened in its first phase, using the returned state for the next wave and possibly receiving a decision value. By the Completeness property of LBV, if a successfully completed LBV instance is elected, then all `wedge&exchange` invocations by correct parties return  $v \neq \perp$  and thus all correct parties decide  $v$  in this wave. Therefore, after a small number of  $\frac{n}{f+1}$  waves all correct parties decide in expectation. Note that the sequence of chosen LBV instances form a properly composed execution, and thus since parties return only values returned from chosen LBVs, our algorithm inherits its safety guarantees from the leader-based protocol the LBV is instantiated with. An illustration of the algorithm appears in Figure 3.

## 6 ACE Instantiation

There are many possible ways to instantiate the ACE framework. We choose to evaluate ACE in the byzantine failure model with  $n = 3f + 1$  parties and a computationally bounded adversary due to the attention it gets in the Blockchain use-case. For the LBV abstraction, we implement a variant of HotStuff [51]. For the leader-election we implement the protocol in [9, 16], and for the Barrier we give an implementation that operates in the same model. All protocols use a BLS threshold signatures schema [14] that requires a setup, which can be done with the help of a trusted dealer or by using a protocol for an asynchronous distributed key generation [33]. Communication is done over TCP to provide reliable links. Due to the space limitation, implementation details can be found in the full paper [49]. The communication complexity of a single LBV is linear and that of the barrier and leader-election is quadratic, leading to an expected total quadratic communication, for each slot.

Our evaluation compares the performance of ACE’s SMR instantiated with HotStuff, we refer to as *ACE HotStuff*, with the base HotStuff SMR implementation. To compare apples to apples, the base HotStuff and ACE HotStuff share as much code as possible. In Section 6.1 we present the tests’ setup. Then, in Section 6.2, we measure ACE’s overhead during failure-free synchronous periods, and in Section 6.3 we demonstrate ACE’s superiority during asynchronous periods and network attacks.

### 6.1 Setup

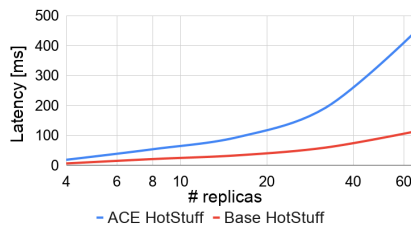
We conducted our experiments using `c5d.4xlarge` instances on AWS EC2 machines in the same data center. We used between 1 and 16 virtual machines, each with 4 replicas. The duration of every test was 60 seconds, and every test was repeated 10 times. The size of the proposed values is 10000 bytes. The latency is measured starting from when a new slot has begun until a decision is made. The throughput is measured in one of two ways. In tests where we altered the number of replicas, the throughput is the total number of bytes committed, divided by the length of the test. In tests where we show the throughput as a function of time, we aggregate the number of committed bytes in 1 second intervals. We did not throttle the bandwidth in any run, rather we altered the transmission delays between the machines, using NetEm [7].

## 6.2 ACE's overhead

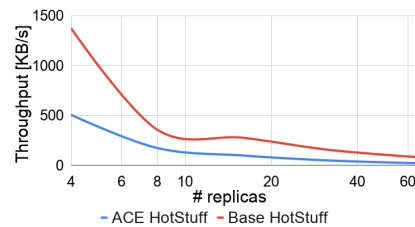
The first set of tests compare ACE HotStuff performance with that of base HotStuff under optimistic, synchronous, faultless conditions. Figure 4 depicts the latency and throughput. The delay on the links was measured to be under  $1ms$ . The latency increases with the growth in the number of replicas since each replica must handle an equal growth in the number of messages. Furthermore, as ACE HotStuff has a larger overhead than base HotStuff, the latency grows faster.

Figure 5 shows the latency and throughput with different delays added to the links. The latency of ACE HotStuff is twice that of base HotStuff. This is expected, as ACE is expected to execute 1.5 waves per slot, leading to 1.5x the latency. Add on the additional barrier, leader election abstraction and we arrive at 2x reduction in performance.

These tests show that the performance cost of using ACE is about 2x reduction in performance in the optimistic case. In the next tests we argue this cost is sometimes worth paying, as liveness of partially synchronous algorithms can be easily affected.

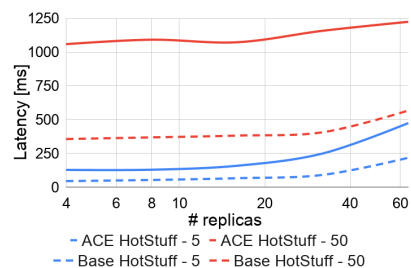


(a) Latency.

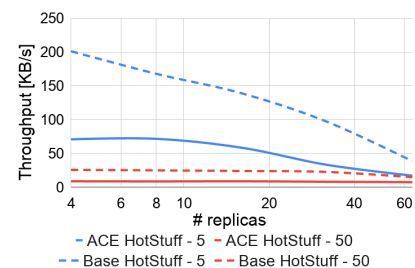


(b) Throughput.

■ **Figure 4** Optimistic case with no network delay.



(a) Latency.



(b) Throughput.

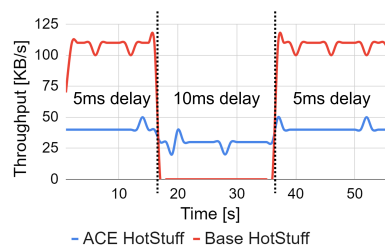
■ **Figure 5** Optimistic case under different network delays.

## 6.3 ACE's superiority

From here on we choose a configuration of 32 replicas and set the transmission delay to be 5ms unless specified otherwise. The second set of tests compare ACE HotStuff and base HotStuff in adverse conditions concerning message delays. These tests manipulate two factors, the transmission delays (controlled via NetEm [7]), and the view timeout strategy.

The first test sets base HotStuff view timers to a fixed constant of 100ms, the time needed for a commit assuming a 5ms transmission delay. The test measures the performance drop during a short period in which transmission delays are increased, simulating asynchrony. For the first third of the test the network delay is 5ms, for the next third the delay is 10ms, and finally the delay returns to 5ms.



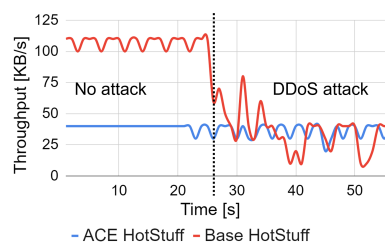


■ **Figure 6** Throughput with a fluctuating transmission delay.

Figure 6 compares the throughput of ACE HotStuff and base HotStuff. While the network delay is 5ms, base HotStuff outperforms ACE HotStuff. However, once the network delay begins to fluctuate, the throughput of base HotStuff goes to 0 since no leader has enough time to drive progress. ACE HotStuff only sees a drop in throughput proportional to the delay, meaning that it continues to progress at network speed.

Note that since the views in base HotStuff are leader-based, byzantine parties (or any other adversarial entity) can achieve the same “asynchronous” effect presented above by only slowing down the leaders. In the next test we demonstrate the above using a *distributed denial of service (DDoS)* attack, in which leaders are flooded with superfluous requests in an attempt to overload them and delay their progress in the leader-based phase.

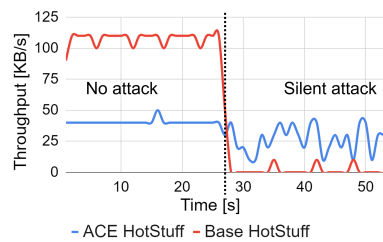
Figure 7 compares the throughput of ACE HotStuff and base HotStuff, where the attack starts at the halfway mark of the test. The byzantine parties coordinate their attack by adaptively choosing a single correct party and flooding it with superfluous requests. In base HotStuff, byzantine parties target correct leaders (byzantine leaders are making progress). In ACE HotStuff, there is no designated leader, thus byzantine parties choose an arbitrary correct party to attack. Our logs show that in base HotStuff progress is mainly made in views where byzantine parties are leaders. If they would not drive progress, the throughput would drop near 0.



■ **Figure 7** Throughput under DDoS attack.

The previous two scenarios operated base HotStuff with a fixed aggressive view timer, which was based on the expected network delay. This caused premature timer expiration during periods of increased delays (due to asynchrony or attacks). One might think that a possible solution can be to set a very long timeouts that will never expire, thus letting the base HotStuff protocol progress in network speed. However, the downside of conservative timers is that byzantine parties can perform a *silent attack* on the protocol’s progress by not driving views when they are leaders, forcing all parties to wait for the long timeouts to expire.

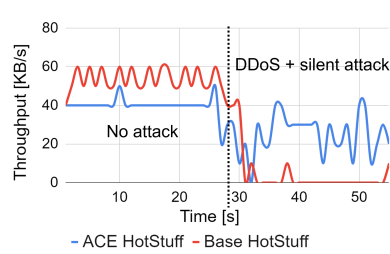




■ **Figure 8** Throughput with conservative timeouts under byzantine silence attack.

The next test evaluates base HotStuff with a conservative view timer of 1 second, fixed to be much higher than expected needed to commit a view, under the silent attack starting at the half way mark. Figure 8 presents the results. Before the attack, base HotStuff indeed progresses in network speed, but during the attack, the throughput drops significantly since a few consecutive byzantine leader might stall progress for seconds. In ACE HotStuff we see a much smaller drop, but more fluctuation. This is due to the fact that byzantine leaders do not drive progress in their LBV instances, and thus the expected number of waves until a decision is now higher.

As the scenarios above demonstrate, neither being too aggressive nor being too conservative works well for base HotStuff during asynchrony or attacks. Therefore, in practice, when HotStuff is deployed it typically adjusts timers during execution according to progress or lack of it. The most common method (used also by PBFT [19] and SBFT [29]) is to increase timeouts whenever timers expires too early, and decrease them whenever progress is made in order to try to learn the network delay and adapt to it's dynamic changes. To test this method, we implement an adaptive version, starting with a delay of  $t$ . If a timeout is reached in a view before a decision is made we set the next view's timeout to  $1.25t$ . Otherwise, the next view's timeout is set to  $0.8t$ .



■ **Figure 9** Throughput with adjusting timeouts under a combination of DDoS and silence attacks.

We evaluate this method against the following attack that combines insights from the previous ones. The results are shown in Figure 9. In the second half of the experiment, byzantine parties perform a DDoS attack on correct leaders, causing the view timers to increase, and then perform the silence attack (in views they act as leaders) to stall progress as much as possible. As expected, base HotStuff throughput drops to almost zero, whereas ACE HotStuff continues driving decisions. Same as in the previous test, ACE HotStuff suffers from fluctuation due to the probability to choose a byzantine leader that did not made progress in its LBV instance. Another interesting phenomenon is the x2 performance drop of base HotStuff before the attack begins compared to previous tests. This is due to the timeout adjustment mechanism, which reduces the timers after every successful view, resulting in a too short timeout in every second view.

While the timer adjustment algorithm can be further enhanced, it is an arms race against the adversary – for each method, there is an adversarial response. In addition, although this evaluation is focused on HotStuff, the only ingredient of the algorithm that is under attack is the timeout, hence the evaluation exemplifies the weakness of all leader-based view by view algorithms. Therefore, our evaluation suggests that the overhead of ACE in the optimistic case is worth paying when high availability is desired under all circumstances.

## 7 Related work

The agreement problem was first introduced by Pease et al. [46], and has since received an enormous amount of attention [17, 8, 50, 19, 34, 41, 39, 10, 20, 11, 42, 40, 23]. One of the most important results is the FLP [27] impossibility, proving that deterministic solutions in the asynchronous communication models are impossible. Below we describe work that was done to circumvent the FLP impossibility, present two related frameworks that were previously proposed for the agreement problem, and discuss alternative fairness definitions. For space limitations, we compare our SMR definition to other systems in the literature in the full paper [49].

**Agreement in the partial synchrony model.** A practical approach to circumvent the FLP impossibility is to consider the partial synchrony communication model [44, 45, 29, 51, 34], which was first proposed by Dwork et al. [25] and later used by seminal works like Paxos [35] and PBFT [19]. As explained in detail in Section 3, protocols designed for this model never violate safety, but provide progress only during long enough synchronous periods. Despite their limitations, they are widely adopted in the industry due to their relative simplicity compared to the alternatives and their performance benefits during synchronous periods. For example, Casandra [1], Zookeeper [2], and Google’s Spanner [26] implement a variant of Paxos [35]; and VMware’s Concord [3], the Libra Network [6] and IBM’s Hyperledger [5], implement SBFT [29], HotStuff [51] and PBFT [19], respectively.

**Agreement in the asynchrony model.** As first shown by Ben-Or [12] and Rabin [37], the FLP impossibility result does not stand randomization. Meaning that the randomized version of the Agreement problem, which guarantees termination with probability 1, can be solved in the asynchronous model provided that parties can flip random coins. The algorithms in [12, 37] are very inefficient in terms of time and message complexity, and there has been a huge effort to improve it over the years. Some considered the theoretical full information model, in which the adversary is computationally unbounded, and showed more efficient algorithms that relax the failure resilience threshold [30, 32]. These are beautiful theoretical results but too complex to implement and maintain.

A more practical model for randomized asynchronous agreement is the random oracle model in which the adversary is computationally bounded and cryptographic assumptions (like the Decisional Diffie–Hellman [22]) are valid. In the context of distributed computing, this model was first proposed by Cachin et al. [16, 15]. In [15] they proposed an almost optimal algorithm for the agreement problem. A variant of this algorithm was later implemented in Honeybadger [42] and Beat [24], which are the first academic asynchronous SMR systems. The protocol in [15] is optimal in terms of resilience to failures and round complexity, but has an inefficient  $O(n^3)$  communication cost. Improving the communication cost was an open problem for almost 20 years, until it was recently resolved in VABA [9]. ACE borrows a lot from VABA [9].

**Frameworks for agreement.** There are a few previously proposed agreement frameworks [28, 36, 18] that we are aware of. The authors of [28] and [18] propose frameworks allowing for dynamically switching between protocols. They observed that no byzantine SMR can outperform all others under all circumstances, and introduce a general way for a system designer to switch between implementations whenever the setting changes. Our work is very different from theirs. While they defined an abstraction in order to compose different SMR view-by-view implementations to achieve better performance in the partially synchronous model, our LBV abstraction provides an API to decouple the leader-based phase from the view-change phase in each view, which in turn allows us to compose LBV instances in a novel way that avoids leader demotions via timeouts and boost liveness in asynchronous networks.

Vertical Paxos [36] is a class of consensus algorithms that separates the mechanism for reaching agreement from the one that deals with failures. The idea is to use a fast and small quorum of parties to drive agreement, and have an auxiliary reconfiguration master to reconfigure this quorum whenever progress stalls. The protocol for agreement relies on the participation of all parties in the dedicated quorum, and thus stalls whenever some party fails. The master is emulated by a bigger quorum, which uses an agreement protocol to agree on reconfiguration, and thus can tolerate failures.

**Fairness.** Although the Agreement and SMR problems have been studied for many years, the question of fairness therein was only recently asked, and we are aware of only few solutions that provide some notion of it [11, 42, 38, 9]. Prime [11] extends PBFT [19] to guarantee that values are committed in a bounded number of slots after they first proposed, and FairLedger [38] uses batching to ensure that all correct party commits a value in every batch. However, in contrast to ACE, both protocols are able to guarantee fairness only during synchronous periods. Honeybadger [42] is an asynchronous protocol that, similarly to FairLedger, batches values proposed by different parties and commits them together atomically. It probabilistically bounds the number of epochs (and accordingly the number of slots) until a value is committed, after being submitted to  $n - f$  parties. The VABA [9] protocol does not use batching, and provides a per slot guarantee that bounds the probability to choose a value proposed by a correct party during asynchronous periods. ACE provides similar fairness guarantees during asynchrony, but also guarantees equal chance for each correct party during synchrony.

## 8 Discussion

In this paper we introduced ACE: a general model agnostic framework for boosting asynchronous liveness of any leader-based SMR system designed for the partially synchronous model. The main ingredient is the novel *LBV* abstraction that encapsulates the properties of a single view in leader-based view-by-view algorithms, while providing an API to control the scheduler of the two phases, leader-based and view-change, in each view. Exploiting this separation, ACE provides a novel algorithm that composes LBV instances in a way that avoids timers and provides a randomized asynchronous SMR solution.

ACE is model agnostic, meaning that it does not add any assumptions on top of what are assumed in the instantiated LBV implementation, thus provides a generic liveness boosting for both byzantine and crash-failure SMRs. In order to instantiate ACE with a specific SMR algorithm, all a system designer needs to do is alter the code of a single view to support LBV's API; this should not be too complicated as the view logic must already implicitly satisfy the required API's properties.

In addition to boosting liveness, ACE is designed in a way that inherently provides fairness due to its randomized election of leaders in retrospect. Moreover, ACE provides a clear separation between safety, which relies on the LBV implementation, and liveness, which is given by the framework. As a result, a system designer that chooses to instantiate ACE gets a modular SMR implementation that is easier to prove correct and maintain – if a better agreement protocol is published, all the designer needs to do in order to integrate it in the system is to alter the LBV implementation accordingly.

To demonstrate the power of ACE we implemented it, instantiated it with the state of the art HotStuff [51] protocol, and compared its performance to the base HotStuff implementation. Our results show that while ACE suffers a 2x performance degradation in the optimistic, synchronous, failure-free case, it enjoys absolute superiority during asynchronous periods and network attacks.

---


## References

- 1 Apache cassandra. <http://cassandra.apache.org/>. Accessed: 2020-05-03.
- 2 Apache zookeeper. <https://zookeeper.apache.org/>. Accessed: 2020-05-03.
- 3 Concord-bft. <https://vmware.github.io/concord-bft/>. Accessed: 2020-05-03.
- 4 etcd: Reliable key-value store. <https://etcd.io/>. Accessed: 2020-05-03.
- 5 Hyperledger. <https://www.hyperledger.org/>. Accessed: 2020-05-03.
- 6 Libra. <https://libra.org/en-US/>. Accessed: 2020-05-03.
- 7 Netem. <https://www.linux.org/docs/man8/tc-netem.html>. Accessed: 2020-05-03.
- 8 Michael Abd-El-Malek, Gregory R Ganger, Garth R Goodson, Michael K Reiter, and Jay J Wylie. Fault-scalable byzantine fault-tolerant services. In *Operating Systems Review*, 2005.
- 9 Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically optimal validated asynchronous byzantine agreement. In *PODC 2019*, New York, NY, USA, 2019. ACM.
- 10 Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. Customizable fault tolerance for wide-area replication. In *Reliable Distributed Systems, 2007. SRDS 2007.*, 2007.
- 11 Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. Prime: Byzantine replication under attack. *IEEE Transactions on Dependable and Secure Computing*, 2011.
- 12 Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *PODC 1983*. ACM, 1983.
- 13 Alysson Bessani, João Sousa, and Eduardo EP Alchieri. State machine replication for the masses with bft-smart. In *DSN 2014*. IEEE, 2014.
- 14 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*. Springer, 2001.
- 15 Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Advances in Cryptology*, 2001.
- 16 Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *Journal of Cryptology*, 18, 2000.
- 17 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*, 1993.
- 18 Carlos Carvalho, Daniel Porto, Luís Rodrigues, Manuel Bravo, and Alysson Bessani. Dynamic adaptation of byzantine consensus protocols. In *SAC 2018*, 2018.
- 19 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *OSDI 1999*, 1999.
- 20 Allen Clement, Edmund L Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *NSDI*, 2009.
- 21 Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. Netpaxos: Consensus at network speed. In *SOSR 2015*, 2015.


- 22 Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976.
- 23 Sisi Duan, Hein Meling, Sean Peisert, and Haibin Zhang. Bchain: Byzantine replication with high throughput and embedded reconfiguration. In Marcos K. Aguilera, Leonardo Querzoni, and Marc Shapiro, editors, *Principles of Distributed Systems*, pages 91–106, Cham, 2014. Springer International Publishing.
- 24 Sisi Duan, Michael K Reiter, and Haibin Zhang. Beat: Asynchronous bft made practical. In *CCS 2018*, 2018.
- 25 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.
- 26 James C. Corbett et al. Spanner: Google’s globally distributed database. *ACM Trans. Comput. Syst.*, 2013.
- 27 Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.
- 28 Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. In *Proceedings of the 5th European conference on Computer systems*, pages 363–376. ACM, 2010.
- 29 Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. Sbft: a scalable and decentralized trust infrastructure. In *DSN 2019*, 2019.
- 30 Bruce M Kapron, David Kempe, Valerie King, Jared Saia, and Vishal Sanwalani. Fast asynchronous byzantine agreement and leader election with full information. *ACM Transactions on Algorithms*, 2010.
- 31 Jonathan Katz and Chiu-Yuen Koo. On expected constant-round protocols for byzantine agreement. In *Annual International Cryptology Conference*, pages 445–462. Springer, 2006.
- 32 Valerie King and Jared Saia. Byzantine agreement in polynomial expected time. In *STOC 2103*. ACM, 2013.
- 33 Eleftherios Kokoris-Kogias, Alexander Spiegelman, Dahlia Malkhi, and Ittai Abraham. Bootstrapping consensus without trusted setup: Fully asynchronous distributed key generation. *IACR Cryptol. ePrint Arch.*, 2019.
- 34 Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *ACM SIGOPS Operating Systems Review*. ACM, 2007.
- 35 Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.
- 36 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, page 312–313, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1582716.1582783.
- 37 Daniel Lehmann and Michael O Rabin. On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem. In *POPL*. ACM, 1981.
- 38 Kfir Lev-Ari, Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi. FairLedger: A Fair Blockchain Protocol for Financial Institutions. In Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller, editors, *23rd International Conference on Principles of Distributed Systems (OPODIS 2019)*, volume 153 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:17, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.OPODIS.2019.4.
- 39 Jinyuan Li and David Mazières. Beyond one-third faulty replicas in byzantine fault tolerant systems. In *NSDI*, 2007.
- 40 Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolić. {XFT}: Practical fault tolerance beyond crashes. In *OSDI 2016*, 2016.
- 41 J-P Martin and Lorenzo Alvisi. Fast byzantine consensus. *DSN 2006*, 2006.

- 42 Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *CCS2 2016*. ACM, 2016.
- 43 Achour Mostéfaoui and Michel Raynal. Signature-free asynchronous byzantine systems: from multivalued to binary consensus with  $t < n/3$ ,  $o(n^2)$  messages, and constant time. *Acta Informatica*, 2017.
- 44 Brian M Oki and Barbara H Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *PODC*. ACM, 1988.
- 45 Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 305–319, 2014.
- 46 Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.
- 47 Michael O Rabin. Randomized byzantine generals. In *SFCS 1983*. IEEE, 1983.
- 48 Nicola Santoro and Peter Widmayer. Time is not a healer. In *STACS 1989*. Springer, 1989.
- 49 Alexander Spiegelman and Arik Rinberg. Ace: Abstract consensus encapsulation for liveness boosting of state machine replication, 2019. [arXiv:1911.10486](https://arxiv.org/abs/1911.10486).
- 50 Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from execution for byzantine fault tolerant services. *SIGOPS Oper. Syst. Rev.*, 37(5):253–267, 2003. doi:10.1145/1165389.945470.
- 51 Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *PODC 2019*. ACM, 2019.


# Security Analysis of Ripple Consensus

**Ignacio Amores-Sesar** 

University of Bern, Switzerland  
ignacio.amores@inf.unibe.ch

**Christian Cachin** 

University of Bern, Switzerland  
christian.cachin@inf.unibe.ch

**Jovana Mičić** 

University of Bern, Switzerland  
jovana.micic@inf.unibe.ch

---

## Abstract

The Ripple network is one of the most prominent blockchain platforms and its native XRP token currently has one of the highest cryptocurrency market capitalizations. The *Ripple consensus protocol* powers this network and is generally considered to a Byzantine fault-tolerant agreement protocol, which can reach consensus in the presence of faulty or malicious nodes. In contrast to traditional Byzantine agreement protocols, there is no global knowledge of all participating nodes in Ripple consensus; instead, each node declares a list of other nodes that it trusts and from which it considers votes.

Previous work has brought up concerns about the liveness and safety of the consensus protocol under the general assumptions stated initially by Ripple, and there is currently no appropriate understanding of its workings and its properties in the literature. This paper closes this gap and makes two contributions. It first provides a detailed, abstract description of the protocol, which has been derived from the source code. Second, the paper points out that the abstract protocol may violate safety and liveness in several simple executions under relatively benign network assumptions.

**2012 ACM Subject Classification** Theory of computation → Cryptographic protocols; Software and its engineering → Distributed systems organizing principles

**Keywords and phrases** Ripple, Blockchain, Quorums, Consensus

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2020.10

**Funding** This work has been funded by the Swiss National Science Foundation (SNSF) under grant agreement Nr. 200021\_188443 (Advanced Consensus Protocols).

## 1 Introduction

Ripple is one of the oldest and most established blockchain networks; its XRP token is ranked fourth in market capitalization in October 2020. The Ripple network is primarily aimed at fast global payments, asset exchange, and settlement. Its distributed consensus protocol is implemented by a peer-to-peer network of validator nodes that maintain a history of all transactions on the network [24]. Unlike Nakamoto’s consensus protocol [21] in Bitcoin or Ethereum, the Ripple consensus protocol does not rely on “mining,” but uses a voting process based on the identities of its validator nodes to reach consensus. This makes Ripple much more efficient than Bitcoin for processing transactions (up to 1500 transactions per second) and lets it achieve very low transaction settlement times (4–5 seconds).

However, Ripple’s consensus protocol does not follow the established models and algorithms for *Byzantine agreement* [22, 15] or *Byzantine fault-tolerant (BFT) consensus* [8]. Those systems start from a common set of nodes that are communicating with each other to reach consensus and the corresponding protocols have been investigated for decades. Instead, the Ripple consensus protocol introduces the idea of subjective validators, such that every



© Ignacio Amores-Sesar, Christian Cachin, and Jovana Mičić;  
licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 10; pp. 10:1–10:16

Leibniz International Proceedings in Informatics



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



## 10:2 Security Analysis of Ripple Consensus

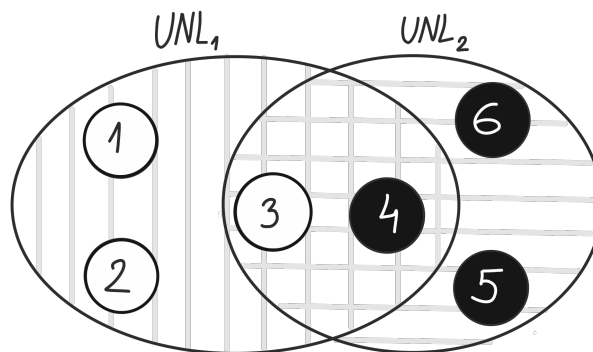
node declares some *trusted validators* and effectively communicates only with those nodes for reaching agreement on transactions. With this mechanism, the designers of Ripple aimed at opening up membership in the set of validator nodes compared to BFT consensus. The trusted validators of a node are defined by a *Unique Node List (UNL)*, which plays an important role in the formalization of the protocol. Every node maintains a static UNL in its configuration file and considers only the opinions of nodes in its UNL during consensus. Figure 1 shows an example network, where two UNLs are defined:  $UNL_1 = \{1, 2, 3, 4\}$  and  $UNL_2 = \{3, 4, 5, 6\}$ ; for instance, nodes 1, 2 and 3 may trust  $UNL_1$ , and nodes 4, 5 and 6 may trust  $UNL_2$ .

Consensus in Ripple aims at delivering the transactions submitted by clients to all participating nodes in a common global order, despite faulty or malicious (Byzantine) nodes [27]. This ensures that the sequence of transactions, which are grouped into so-called *ledgers* and then processed by each node, is the same for all nodes. Hence, the states of all correct nodes remain synchronized, according to the blueprint of state-machine replication [26].

Cachin and Vukolić [7] have earlier pointed out that it is important to formally assess the properties of blockchain consensus protocols. Unfortunately, many systems have been designed and were deployed without following the agreed-on principles on protocol analysis from the literature. Ripple is no exception to this, as we show in this work.

Specifically, we focus on two properties that every sound protocol must satisfy [1]: *safety* and *liveness*. Safety means that nothing “bad” will ever happen, and liveness means that something “good” eventually happens. Safety ensures that the network does not fork or double-spend a token, for instance. A violation of liveness would mean that the network stops making progress and halts processing transactions, which creates as much harm as forking.

This work first presents a complete, abstract description of the Ripple consensus protocol (Section 3). The model has been obtained directly from the source code. It is formulated in the language spoken by designers of consensus protocols, in order to facilitate a better understanding of the properties of Ripple consensus. No formal description of Ripple consensus with comparable technical depth has been available so far (apart from the source itself).



■ **Figure 1** Example of a Ripple network configuration with six nodes and two UNLs,  $UNL_1 = \{1, 2, 3, 4\}$  and  $UNL_2 = \{3, 4, 5, 6\}$ . Nodes 1, 2, and 3 (white) trust  $UNL_1$ , and nodes 4, 5, and 6 (black) trust  $UNL_2$ . Notice that nodes 3 and 4 have more influence than the rest of nodes since they are in the intersection of both UNLs.



Second, we exhibit examples of how safety and liveness may be violated in executions of the Ripple consensus protocol (Sections 4 and 5). In particular, the *network may fork* under the standard condition on UNL overlap stated by Ripple and in the presence of a constant *fraction* of Byzantine nodes. The malicious nodes may simply send conflicting messages to correct nodes and delay the reception of other messages among correct nodes. Furthermore, the consensus protocol may *lose liveness* even if all nodes have the same UNL and there is only *one* Byzantine node. If this would occur, the system has to be restarted manually.

Given these findings, we conclude that the consensus protocol of the Ripple network is brittle and does not ensure consensus in the usual sense. It relies heavily on synchronized clocks, timely message delivery, the presence of a fault-free network, and an a-priori agreement on common trusted nodes. The role of the UNLs, their overlap, and the creation of global consensus from subjective trust choices remain unclear. If Ripple instead had adopted a standard BFT consensus protocol [5], as done by Tendermint [4], versions of Hyperledger Fabric [2], Libra [16] or Concord [13], then the Ripple network would resist a much wider range of corruptions, tolerate temporary loss of connectivity, and continue operating despite loss of synchronization.

## 2 Related work

Despite Ripple’s prominence and its relatively high age among blockchain protocols – the system was first released in 2012 – there are only few research papers investigating the Ripple consensus protocol compared to the large number of papers on Bitcoin. The original Ripple white paper of 2014 [27] describes the UNL model and illustrates some ideas behind the protocol. It claims that under the assumption of requiring an 80%-quorum for declaring consensus, the intersection between the UNLs of any two nodes  $u$  and  $v$  should be larger than 20% of the size of the larger of their UNLs, i.e.,

$$|UNL_u \cap UNL_v| \geq \frac{1}{5} \max\{|UNL_u|, |UNL_v|\}.$$

The only earlier protocol analysis in the scientific literature of which we are aware was authored by Armknecht et al. in 2015 [3]. This work analyzes the Ripple consensus protocol and outlines the security and privacy of the network compared to Bitcoin. The authors prove that a 20%-overlap, as claimed in the white paper, cannot be sufficient for reaching consensus and they increase the bound on the overlap to at least 40%, i.e.,

$$|UNL_u \cap UNL_v| > \frac{2}{5} \max\{|UNL_u|, |UNL_v|\}$$

In a preprint of 2018, Chase and MacBrough [10] further strengthen the required UNL overlap. They introduce a high-level model of the consensus protocol and describe some of its properties, but many details appear unclear or are left out. This work concludes that the overlap between UNLs should actually be larger than 90%. The paper also gives an example with 102 nodes that shows how liveness can be violated, even if the UNLs overlap almost completely (by 99%) and there are no faulty nodes. The authors conclude that manual intervention would be needed to resurrect the protocol after this.

Christodoulou et al. [11] further investigate the decentralization of a Ripple network by running simulations with different configurations. They observe how the convergence time and the so-called “Network Health Indicator,” a synthetic measure computed by a tool available from Ripple, change depending on the overlap between the nodes’ UNLs. Their experiments suggest that a large UNL overlap is required only with more than 20% of malicious nodes in the network. They also indicate a possibility for a dynamic determination of an optimal UNL overlap.

An analysis whose goal is similar to that of our work has been conducted by Mauri et al. [19]. Based on the source code, they give a verbal description of the consensus protocol, but do not analyze dynamic protocol properties. Our analysis, in contrast, provides a detailed, formal description with pseudocode and achieves a much better understanding of how the “preferred ledger” is chosen. Moreover, our work shows possible violations of safety and liveness, whereas Mauri et al. address only on the safety of the consensus protocol through sufficient conditions.

Other academic work mostly addresses network structure, transaction graph, and privacy aspects of payments on the Ripple blockchain [18, 20], which is orthogonal to our focus.

### 3 A description of the Ripple consensus protocol

The main part of our analysis consists of a detailed presentation of the Ripple consensus protocol in this section and formally in Algorithms 1–3. Before we describe this, we define the task that the protocol intends to solve.

#### 3.1 Specification

Informally, the goal of the Ripple consensus protocol is “to ensure that the same transactions are processed and validated ledgers are consistent across the peer-to-peer XRP Ledger network” [25]. More precisely, this protocol implements the task of synchronizing the nodes so that they proceed through a common execution, by appending successive *ledgers* to an initially empty history and where each ledger consists of a number of *transactions*. This is the problem of replicating a service in a distributed system, which goes back to Lamport et al.’s pioneering work on Byzantine agreement [22, 15]. The problem has a long history and a good summary can be found in the book “30-year perspective on replication” [9].

For replicating an abstract service among a set of nodes, the service is formulated as a deterministic state machine that executes *transactions* submitted by clients or, for simplicity, by the nodes themselves. The *consensus protocol* disseminates the transactions among the nodes, such that each node locally executes the *same sequence of transactions* on its copy of the state. The task provided by this protocol is also called *atomic broadcast*, indicating that the nodes actually disseminate the transactions. When each node locally executes the same sequence of transactions, as directed by the protocol, and since each transaction is deterministic, all nodes will maintain the same copy of the state [26].

More formally, *atomic broadcast* is characterized by two events dealing with transactions: *submission* and *execution*, which may each occur multiple times. Every node may submit a transaction  $tx$  by invoking  $submit(tx)$  and atomic broadcast applies  $tx$  to the application state on the node through  $execute(tx)$ . A protocol for atomic broadcast then ensures these properties [14, 5]:

**Validity:** If a correct node  $p$  submits a transaction  $tx$ , then  $p$  eventually executes  $tx$ .

**Agreement:** If a transaction  $tx$  is executed by some correct node, then  $tx$  is eventually executed by every correct node.

**Integrity:** No correct node executes a transaction more than once; moreover, if a correct node executes a transaction  $tx$  and the submitter  $p$  of  $tx$  is correct, then  $tx$  was previously submitted by  $p$ .

**Total order:** For transactions  $tx$  and  $tx'$ , suppose  $p$  and  $q$  are two correct nodes that both execute  $tx$  and  $tx'$ . Then  $p$  executes  $tx$  before  $tx'$  if and only if  $q$  executes  $tx$  before  $tx'$ .

Our specification does not refer to the heterogeneous trust structure defined by the UNLs and simply assumes all nodes should execute the same transactions. This corresponds to the implicit assumption in Ripple’s code and documentation. We note that the question of establishing global consistency in a distributed system with subjective trust structures is a topic of current research, as addressed by asymmetric quorum systems [6] or in the context of Stellar’s protocol [17], for example.

### 3.2 Overview

The following description was obtained directly from the source code. Its overall structure retains many elements and function names found in the code, so that it may serve as a guide to the source for others and to explain its working. If the goal had been to compare Ripple consensus to the existing literature on synchronous Byzantine agreement protocols, the formalization would differ considerably.

The protocol is highly *synchronous* and relies on a common notion of time. It is structured into successive *rounds of consensus*, whereby each round agrees on a *ledger* (a set of transactions to execute). Each round roughly takes a predefined amount of time and is driven by a heartbeat timer, which triggers a state update once per second. This contrasts with the Byzantine consensus protocols with partial synchrony [12], such as PBFT [8], which can tolerate arbitrarily long periods of asynchrony and rely on clocks or timeouts only for liveness. The Ripple protocol aims to agree on a transaction set within each synchronized round. The round ends when all nodes collectively declare to have reached consensus on a *proposal* for the round. The protocol is then said to *close* and later *validate* a ledger containing the agreed-on transaction set. However, the transactions in the ledger are executed only after another protocol step, once the ledger has become *fully validated*; this occurs in an asynchronous process in the background. Transaction execution is only logically synchronized with the consensus round.

A *ledger* consists of a batch of transactions that result from a consensus round and contains a hash of the logically preceding ledger. Ledgers are stored persistently and roughly play the role of blocks in other blockchain protocols. Each node locally maintains three different ledgers: the *current ledger*, which is in the process of building during a consensus round, the *previous ledger*, representing the most recently closed ledger and the *valid ledger*, which is the last fully validated ledger in the network.

In more detail, a consensus round has three *phases*: *open*, *establish*, and *accepted*. The usual phase transition goes from *open* to *establish* to *accepted* and then proceeds to the next consensus round, which starts again from *open*. However, it is also possible that the phase changes from *establish* to *open*, if a node detects that it has been forked from the others to a wrong ledger and resumes processing after switching to the ledger agreed by the network.

Nodes may submit transactions at any time, concurrently to executing the consensus rounds. They are disseminated among the nodes through a *gossip layer* that ensures only weak consistency. All transactions that have been received from gossip are placed into a buffer. Apparently, the original design assumed that the gossip layer ensures a notion of consistency that prevents Byzantine nodes from equivocating, in the sense of correct nodes never receive different messages from them. This assumption has been dropped later [10].

The protocol rounds and their phases are implemented by a state machine, which is invoked every second, when the global *heartbeat* timer ticks. Messages from other nodes are received asynchronously in the background and processed during the next timer interrupt.

The timeout handler (L56) first checks if the local *previous ledger* is the same as the *preferred ledger* of a sufficient majority of the nodes in the network. If not, the node has been forked or lost synchronization with the rest of the network and must bring itself back to the state agreed by the network. In this case, it starts a new consensus round from scratch.

When the node enters a new round of consensus, it sets the phase to *open*, resets round-specific data structures, and simply waits for the buffer to fill up with submitted transactions. Once the node has been in the *open* phase for more than half of the duration of the previous consensus round, the node moves to the *establish* phase (L63–L64; function *closeLedger*). It locally closes the ledger, which means to initialize its proposal for the consensus round and to send this to the other nodes in its UNL.

During the *establish* phase, the nodes exchange their proposals for the transactions to decide in this consensus round (using PROPOSAL messages). Obviously, these proposals may contain different transaction sets. All transactions on which the proposals from other nodes differ become *disputed*. Every node keeps track of how many other nodes in its UNL have proposed a disputed transaction and represents this information as *votes* by the other nodes. The node may remove a disputed transaction from its own proposal, or add one to its proposal, based on the votes of the others and based on the time that has passed. Specifically, the node increases the necessary threshold of votes for changing its own vote on a disputed transaction depending on the duration of the *establish* phase with respect to the time taken by the previous consensus round.

The node leaves the *establish* phase when it has found that there is a consensus on its proposal (L69–L71; functions *haveConsensus* and *onAccept*). The node constructs the next ledger (the “last closed ledger”) by “applying” the decided transactions. This ledger is signed and broadcast to the other nodes in a VALIDATION message.

The node then moves to the *accepted* phase and immediately initializes a new consensus round. Concurrently, the node receives VALIDATION messages from the nodes in its UNL. It verifies them and counts how many other nodes in its UNL have issued the same validation. When this number reaches 80% of the nodes in its UNL, the ledger becomes fully validated and the node executes the transactions contained in it.

### 3.3 Details

**Functions.** For simplicity, there are some functions that are not fully explained in the pseudocode. These functions are:

- *startTimer(timer, duration)* starts *timer*, which expires after the time passed as *duration*.
- *clock.now()* returns the current time.
- *Hash()* creates a unique identifier (often denoted *ID*) of a data structure by converting the data to a canonical representation and applying a cryptographic hash function.
- $A \triangle B$  denotes the symmetric set difference.
- *boolToInt(b)* converts a logical value *b* to an integer and returns *b*? 0:1.
- *sign<sub>i</sub>(L)* creates a cryptographic digital signature for ledger *L* by node *i*.
- *verify<sub>i</sub>(L, σ)* checks if the digital signature on *L* from node *i* is valid.
- *siblings(M)* returns the set of nodes, different from *M*, that have the same parent as *M*.

**Remarks on the pseudocode.** Next to every function name, a comment points to a specific file and line in the source code which contains its implementation. The Ripple source contains a large number of files and most of the consensus protocol implementation is actually spread

over multiple header (.h) files, which complicates the analysis of the code. The references in this work are based on version 1.4.0<sup>1</sup> of `rippled` [23].

**Phase open.** Function `beginConsensus` starts a consensus round for the next ledger (L50). Each ledger (L11) contains a hash (`ID`) that serves as its identifier, a sequence number (`seq`), a hash of the parent ledger (`parentID`), and a transaction set (`txns`), denoting the transactions applied by the ledger.

The node records the time when the `open` phase started (`openTime`, L54), so that it can later calculate how long the `open` phase has taken. This is important because the duration of the `open` phase determines when to close the ledger locally. If the time that has passed since `openTime` is longer or equal to half of the previous round time (`prevRoundTime`), consensus moves to phase `establish` by calling the function `closeLedger` (L64). Meanwhile all nodes submit transactions with the gossip layer (L46) and each node stores the transaction received via gossip messages in its transaction set  $S$  (L48). We model transactions as bit strings. In some places, and as in the source code, we use a short, unique *transaction identifier* (of type `int`) for each transaction  $tx \in \{0, 1\}^*$ , computed by a function `TxID(tx)`. A transaction set is a set of binary strings here, but the source code maintains a transaction set using a hash map, containing the transaction data indexed by their identifiers.

**Phase establish.** When the node moves from `open` to `establish`, it calls `closeLedger` that creates an initial proposal (stored in `result.proposal`), containing all transactions received from the gossip layer (L79) that have not been executed yet. A proposal structure (L16) contains the hash of the previous ledger (`prevLedgerID`), a sequence number (`seq`), the actual set (`txns`) of proposed transactions (in the source code named `position`), an identifier of the node (`node`) that created this proposal, and a timestamp (`time`) when this proposal is created (L79).

The node then broadcasts the new proposal as a `PROPOSAL` message (L81) to all nodes in its UNL. When they receive it, they will store its contents in their `currPeerProposals` collection of proposals (L85), if the message originates from a node their respective UNL. The `closeLedger` function also sets `result.roundTime` to the current time (L80). This serves to measure the duration of the `establish` phase and will be used later to determine how far the consensus process has converged.

Based on the proposals from other nodes, each node computes a set of disputed transactions (L88). A disputed transaction (`DisputedTx`, L5) contains the transaction itself (`tx`), a binary vote (`ourVote`) by the node on whether this transaction should be included in the ledger, the number of “yes” and “no” votes from other nodes on the transaction (`yays` and `nays`), taken from their `PROPOSAL` messages, and the list of votes on this transaction from the other nodes (`votes`).

A transaction becomes disputed when it is proposed by the node itself and some other node does not propose it, or vice versa. The node determines these by comparing its own transaction set with the transaction sets of all other nodes (L89). Every disputed transaction is recorded (as a `DisputedTx` structure) in the collection `result.disputes` (L90–L97).

During the `establish` phase, the node constantly updates its votes on all disputed transactions (L68; L99; L117) for responding to further `PROPOSAL` messages that have been received. A vote may change based on the number of nodes in favor of the transaction, the *convergence*

<sup>1</sup> The latest release (16. November 2020) is version 1.6.0. Compared to version 1.4.0, the current release has no significant changes concerning the consensus protocol.

*ratio* (*converge*) and a *threshold*. Convergence measures the expected progress in one single consensus round and is computed from the duration of the *establish* phase, the duration of the previous round, and an assumed maximal consensus-round time (L67). The value for the threshold is predefined. The further the consensus converges, the higher is the threshold that the number of opposing votes needs to reach so that the node changes its own vote (L126). Whenever the node’s proposal is updated, the node broadcasts its new proposal to the other nodes (L113) and the disputed transactions are recomputed (L115).

Afterwards, the node checks if consensus on its proposed transaction set *result.txns* is reached, by calling the function *haveConsensus* (L69). The node counts agreements (L130) and disagreements (L131) with *result.txns*. If the fraction of agreeing nodes is at least 80% with respect to the UNL (L132), then consensus is reached. The node proceeds to the *accepted* phase by calling the function *onAccept* (L71).

**Phase accepted.** The function *onAccept* (L133) “applies” the agreed-on transaction set and thereby creates the next ledger (called the “last closed ledger” in the source code; L134). This ledger is then signed (L136) and broadcast to the other nodes as a VALIDATION message (L137). This marks the end of the *accepted* phase and a new consensus round is initiated by the node (L140).

Meanwhile, in the background, the node receives VALIDATION messages from other nodes in its UNL and tries to verify them (L141). This verification checks the signature and if the sequence number of the received ledger is the same as the sequence number of the own ledger. All validations that satisfy both conditions and contain the node’s own agreed-on ledger are counted (L145); this comparison uses the cryptographic hash of the ledger structure in the source code. Again, if 80% of nodes have validated the same ledger and if the sequence number of that ledger is larger than that of the last fully validated ledger (L146), the ledger becomes fully validated (L147). The node then executes the transactions in the ledger (L150). In other words, the consensus decision has become final.

**Preferred ledger.** A node participating in consensus regularly computes the *preferred ledger*, which denotes the current ledger on which the network has decided. Due to possible faults and network delays, the node’s *prevLedger* may have diverged from the preferred ledger, which is determined by calling the function *getPreferred(validLedger)* (L151). Should the network have adopted a different ledger than the *prevLedger* of the node, the node switches to this ledger and restarts the consensus round with the new ledger.

Notice that the validated ledgers from all correct nodes form a tree, rooted in the initial ledger (*genesisLedger*). Each node stores all valid ledgers that it receives in a tree-structured variable *tree*. Whenever the node receives a VALIDATION message containing a ledger  $L'$ , it adds  $L'$  to *tree* (L143). In order to compute the preferred ledger, we define the following functions, which are derived from the ledgers in *tree* and in the received VALIDATION messages:

- *tip-support*( $L$ ) for a ledger  $L$  is the number of validators in the UNL that have validated  $L$ . In other words,

$$\text{tip-support}(L) = |\{ j \in \text{UNL} \mid \text{validations}[j] = L \}|.$$

- *support*( $L$ ) for a given ledger  $L$  is the sum of the tip support of  $L$  and all its descendants in *tree*, i.e.,

$$\text{support}(L) = \text{tip-support}(L) + \sum_{L' \text{ is a child of } L \text{ in } \text{tree}} \text{tip-support}(L').$$



---

**Algorithm 1** Ripple consensus protocol for node  $i$  (continues on next pages).
 

---

```

1: Type
2:   Enum Phase = { open, establish, accepted }
3:   Tx = {0, 1}* // a transaction
4:   TxSet = 2Tx
5:   DisputedTx( // DisputedTx.h:50
6:     Tx tx, // disputed transaction
7:     bool ourVote, // binary vote on whether transaction should be included
8:     int yays, // number of yes votes from others
9:     int nays, // number of no votes from others
10:    HashMap[int → bool] votes) // collection of votes indexed by node
11:   Ledger( // Ledger.h:77
12:     Hash ID, // identifier
13:     int seq, // sequence number of this ledger
14:     Hash parentID, // identifier of ledger's parent
15:     TxSet txns) // set of transactions applied by ledger
16:   Proposal( // ConsensusProposal.h:52
17:     Hash prevLedgerID, // hash of the previous ledger, on which this proposal builds
18:     int seq, // sequence number
19:     TxSet txns, // proposed transaction set, called position at ConsensusProposal.h:73
20:     int node, // node that proposes this
21:     milliseconds time) // time when proposal is created
22:   ConsensusResult( // ConsensusTypes.h:201
23:     TxSet txns, // set of transactions consensus agrees on
24:     Proposal proposal, // proposal containing transaction set
25:     HashMap[int → DisputedTx] disputes, // collection of disputed transactions
26:     milliseconds roundTime) // duration of the establish phase

27: State
28:   Phase phase // phase of the consensus round for agreeing on one ledger
29:   Tree tree // tree representation of received valid ledgers
30:   Ledger L // current working ledger
31:   Ledger prevLedger // last agreed-on ("closed") ledger according to the network
32:   Ledger validLedger // ledger that was most recently fully validated by the node
33:   TxSet S // transactions submitted by clients that have not yet been executed
34:   ConsensusResult result // data relevant for the outcome of consensus on a single ledger
35:   HashMap[int → Proposal] currPeerProposals // collection of proposals indexed by node
36:   HashMap[int → Ledger] validations // collection of validations indexed by node
37:   milliseconds prevRoundTime // time taken by the previous consensus round, initialized to 15s
38:   float converge ∈ [0, 1] // ratio of round time to prevRoundTime
39:   UNL ⊆ {1, ..., M} // validator nodes trusted by node  $i$ , taken from the configuration file
40:   milliseconds openTime // time when the last open phase started

41: function initialization()
42:   prevLedger ← genesisLedger // genesisLedger is the first ledger in the history of the network
43:   S ← {}
44:   beginConsensus() // start the first round of consensus
45:   startTimer(heartbeat, 1s) // NetworkOPs.cpp:673

46: upon submission of a transaction  $tx$  do
47:   send message [SUBMIT,  $tx$ ] with the gossip layer

48: upon receiving a message [SUBMIT,  $tx$ ] from the gossip layer do
49:    $S \leftarrow S \cup \{tx\}$ 

50: function beginConsensus() // start a new round of consensus, Consensus.h:663
51:   phase ← open // Consensus.h:669
52:   result ← ({}, ⊥, [], 0) // Consensus.h:674
53:   converge ← 0 // Consensus.h:675
54:   openTime ← clock.now() // remember the time when this consensus round started
55:   currPeerProposals ← [] // reset the proposals for this consensus round

```

---

---

**Algorithm 2** Ripple consensus protocol for node  $i$  (continued).

---

```

56: upon timeout(heartbeat) do // Consensus.h:818
57:    $L' \leftarrow \text{getPreferred}(\text{validLedger})$ 
58:   if  $L' \neq \text{prevLedger}$  then
59:      $\text{prevLedger} \leftarrow L'$ 
60:      $\text{beginConsensus}(\text{prevLedger})$ 
61:   if  $\text{phase} = \text{open}$  then // wait until the closing ledger can be determined locally
62:     if  $(\text{clock.now}() - \text{openTime}) \geq \frac{\text{prevRoundTime}}{2}$  then // Consensus.cpp:75
63:        $\text{phase} \leftarrow \text{establish}$ 
64:        $\text{closeLedger}()$  // initialize consensus value in  $\text{result}$ 
65:     else if  $\text{phase} = \text{establish}$  then // agree on the contents of the ledger to close
66:        $\text{result.roundTime} \leftarrow \text{clock.now}() - \text{result.roundTime}$ 
67:        $\text{converge} \leftarrow \frac{\text{result.roundTime}}{\max\{\text{prevRoundTime}, 5s\}}$ 
68:        $\text{updateOurProposals}()$  // update consensus value in  $\text{result}$ 
69:       if  $\text{haveConsensus}()$  then
70:          $\text{phase} \leftarrow \text{accepted}$ 
71:          $\text{onAccept}()$  // note this immediately sets  $\text{phase} = \text{open}$  inside  $\text{beginConsensus}()$ 
72:       else if  $\text{phase} = \text{accepted}$  then // Consensus.h:821
73:         // do nothing
74:          $\text{startTimer}(\text{heartbeat}, 1s)$ 

75: // transition from  $\text{open}$  to  $\text{establish}$  phase
76: function  $\text{closeLedger}()$  // Consensus.h:1309
77:    $L \leftarrow (\perp, \text{prevLedger.seq} + 1, \perp, \{\})$ 
78:    $\text{result.txns} \leftarrow S$  // propose the current set of submitted transactions
79:    $\text{result.proposal} \leftarrow (\text{Hash}(\text{prevLedger}), 0, \text{result.txns}, i, \text{clock.now}())$ 
80:    $\text{result.roundTime} \leftarrow \text{clock.now}()$ 
81:   broadcast message [PROPOSAL,  $\text{result.proposal}$ ]
82:    $\text{result.disputes} \leftarrow []$  // disputes for transactions not proposed by all nodes in the UNL
83:   for  $j \in \text{UNL}$  such that  $\text{currPeerProposals}[j] \neq \perp$  do
84:      $\text{createDisputes}(\text{currPeerProposals}[j].\text{txns})$  // compared to  $\text{result.txns}$ , Consensus.h:1334

85: upon receiving a message [PROPOSAL,  $\text{prop}$ ] such that  $\text{prop} = (nl, \cdot, \cdot, j, \cdot)$  and
86:    $j \in \text{UNL}$  and  $nl = \text{Hash}(\text{prevLedger})$  do
87:    $\text{currPeerProposals}[j] \leftarrow \text{prop}$  // Consensus.h:781

88: function  $\text{createDisputes}(\text{TxSet set})$  // Consensus.h:1623
89:   for  $tx \in \text{result.txns} \Delta \text{set}$  do // all transactions that differ between  $\text{result.txns}$  and  $\text{set}$ 
90:      $dt \leftarrow (tx, (tx \in \text{result.txns}), 0, 0, [])$  //  $dt$  is a disputed transaction
91:     for  $k \in \text{UNL}$  such that  $\text{currPeerProposals}[k] \neq \perp$  do
92:       if  $tx \in \text{currPeerProposals}[k].\text{txns}$  then
93:          $dt.\text{votes}[k] \leftarrow 1$  // record node's vote for the disputed transaction
94:          $dt.\text{yays} \leftarrow dt.\text{yays} + 1$ 
95:       else
96:          $dt.\text{votes}[k] \leftarrow 0$  // record node's vote against the disputed transaction
97:          $dt.\text{nays} \leftarrow dt.\text{nays} + 1$ 
98:      $\text{result.disputes}[\text{TxID}(tx)] \leftarrow dt$  // phase  $\text{establish}$ 

99: function  $\text{updateOurProposals}()$  // Consensus.h:1361
100:   for  $j \in \text{UNL}$  such that  $(\text{clock.now}() - \text{currPeerProposals}[j].\text{time}) > 20s$  do
101:      $\text{currPeerProposals}[j] \leftarrow \perp$  // remove stale proposals
102:    $T \leftarrow \text{result.txns}$  // current set of transactions, to update from disputed ones
103:   for  $dt \in \text{result.disputes}$  do //  $dt$  is a disputed transaction
104:     if  $\text{updateVote}(dt)$  then // if vote on  $dt$  changes, update the dispute set
105:        $dt.\text{ourVote} \leftarrow \neg dt.\text{ourVote}$ 
106:       if  $dt.\text{ourVote}$  then // should the transaction be included? DisputedTx.h:77
107:          $T \leftarrow T \cup \{dt.tx\}$  //  $dt.\text{ourVote}$  is initially set in  $\text{createDisputes}(\text{TxSet set})$ 
108:       else
109:          $T \leftarrow T \setminus \{dt.tx\}$ 
110:   if  $T \neq \text{result.txns}$  then // if  $\text{txns}$  changed, then update  $\text{result}$  and tell the other nodes
111:      $\text{result.txns} \leftarrow T$ 
112:      $\text{result.proposal} \leftarrow (\text{Hash}(\text{prevLedger}), \text{result.proposal.seq} + 1, \text{result.txns}, i)$ 
113:     broadcast message [PROPOSAL,  $\text{result.proposal}$ ]
114:      $\text{result.disputes} \leftarrow []$  // recompute  $\text{disputes}$  after updating  $\text{result.txns}$ 
115:     for  $j \in \text{UNL}$  such that  $\text{currPeerProposals}[j] \neq \perp$  do // Consensus.h:1679
116:        $\text{createDisputes}(\text{currPeerProposals}[j].\text{txns})$ 

```

---



■ **Algorithm 3** Ripple consensus protocol for node  $i$  (continued).

---

```

117: function updateVote(DisputedTx dt) // DisputedTx.h:197
118:   if converge < 0.5 then // set threshold based on duration of the establish phase
119:     threshold ← 0.5
120:   else if converge < 0.85 then
121:     threshold ← 0.65
122:   else if converge < 2 then
123:     threshold ← 0.7
124:   else
125:     threshold ← 0.95
126:   newVote ←  $\left( \frac{dt.yays + \text{boolToInt}(dt.ourVote)}{dt.yays + dt.nays + 1} > \text{threshold} \right)$ 
127:   return (newVote ≠ dt.ourVote) // the vote changes

128: function haveConsensus() // Consensus.h:1545
129:   // count number of agreements and disagreements with our proposal
130:   agree ←  $|\{j | \text{currPeerProposals}[j] = \text{result.proposal}\}|$ 
131:   disagree ←  $|\{j | \text{currPeerProposals}[j] \neq \perp \wedge \text{currPeerProposals}[j] \neq \text{result.proposal}\}|$ 
132:   return  $\left( \frac{\text{agree} + 1}{\text{agree} + \text{disagree} + 1} \geq 0.8 \right)$  // 0.8 is defined in ConsensusParams.h, Consensus.cpp:104
   // phase accepted

133: function onAccept() // RCLConsensus.cpp:408
134:   L ← (prevLedger, result.txns) // L is the last closed ledger, RCLConsensus.cpp:708
135:   validations[i] ← L
136:   σ ← signi(L) // validate the ledger, RCLConsensus.cpp:743
137:   broadcast message [VALIDATION, i, σ, L]
138:   prevLedger ← L // store the last closed ledger
139:   prevRoundTime ← result.roundTime
140:   beginConsensus() // advance to the next round of consensus, NetworkOPs.cpp:1584

141: upon receiving a message [VALIDATION, j, σ, L'] such that // LedgerMaster.cpp:858
142:   L'.seq = L.seq and verifyj(L', σ) do
143:   add L' to tree
144:   validations[j] ← L' // store received validation
145:   valCount ←  $|\{k \in \text{UNL} | \text{validations}[k] = L'\}|$  // count the number of validations
146:   if valCount ≥ 0.8 · |UNL| and L.seq > validLedger.seq then
147:     validLedger ← L // ledger becomes fully validated
148:     S ← S ∪ {L.txns}
149:     for tx ∈ L.txns do // in some deterministic order
150:       execute(tx)

151: function getPreferred(Ledger L) // LedgerTrie.h:677
152:   if L is a leaf node in tree then
153:     return L
154:   else
155:     M ← arg max{support(N) | N is a child of L in the tree}
156:     if uncommitted(M) ≥ support(M) then
157:       return L
158:     else if max{support(N) | N ∈ siblings(M)} + uncommitted(M) < support(M) then
159:       return getPreferred(M)
160:     else
161:       return L

```

---

## 10:12 Security Analysis of Ripple Consensus

- $uncommitted(L)$  for a ledger  $L$  denotes the number of validators whose last validated ledger has a sequence number that is strictly smaller than the sequence number of  $L$ . More formally,

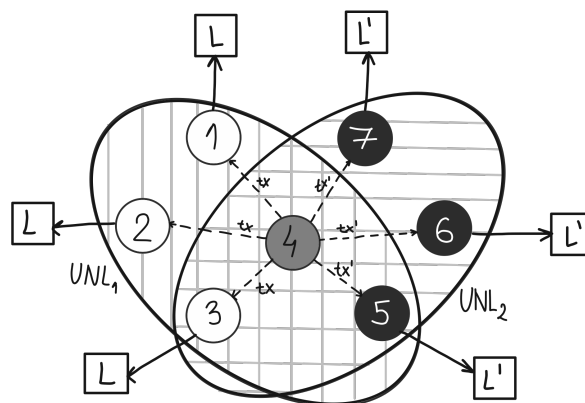
$$uncommitted(L) = |\{j \in UNL \mid validations[j].seq < L.seq\}|.$$

With these definitions, we now explain how  $getPreferred(Ledger L)$  proceeds (L151–L161). If  $L$  has no children in  $tree$ , it returns  $L$  itself. Otherwise, the function considers the child of  $L$  that has the highest support among all children ( $M$ ). If the support of  $M$  is still smaller than the number of validators that are yet uncommitted at this ledger-sequence number, then  $L$  is still the preferred ledger (L157). Otherwise, if the support of  $M$  is guaranteed to exceed the support of any of its siblings  $N$ , even when the uncommitted validators would also support  $N$ , then the function recursively calls  $getPreferred$  on  $M$ , which outputs the preferred ledger for  $M$  and returns this as the preferred ledger for  $L$ . Otherwise,  $L$  itself is returned as the preferred ledger. Observe that in the case when  $M$  has no siblings conditions in L156 and L158 are equivalent. Then is enough to check if support of  $M$  is greater than uncommitted od  $M$ .

### 4 Violation of safety

In this section, we address the safety of the Ripple consensus protocol. We describe a simple scenario that violates consensus in an execution with seven nodes, of which one is Byzantine. Actually, one can generalize this to executions with more nodes.

To show that the Ripple consensus protocol violates safety and may let two correct nodes execute different transactions, we use the following scenario with seven nodes. Figure 2 gives a graphical representation of our scenario. Nodes are named by numbers. We let  $UNL_1 = \{1, 2, 3, 4, 5\}$  and  $UNL_2 = \{3, 4, 5, 6, 7\}$ , as illustrated by the two hatched areas in the figure. Nodes 1, 2, and 3 (white) trust  $UNL_1$ , nodes 5, 6, and 7 (black) trust  $UNL_2$ , and they are all correct; node 4 (gray) is Byzantine. With this setup, we achieve 60% overlap between the UNLs of any two nodes.



■ **Figure 2** Example setup for showing a safety violation in the Ripple consensus protocol. The setup consists of seven nodes, one of them Byzantine, and two UNLs. Nodes 1, 2, and 3 (white) adopt  $UNL_1$ , vertically hatched, and nodes 5, 6, and 7 adopt  $UNL_2$ , horizontally hatched. Node 4 (gray) is Byzantine.

The key idea is that the Byzantine node (4) changes its behavior depending on the group of nodes to which it communicates. It will cause nodes 1, 2, and 3 (white) to propose some transaction  $tx$  and nodes 5, 6, and 7 (black) to propose a transaction  $tx'$  for the next ledger. No other transaction exists. The Byzantine node (4) follows the protocol as if it had proposed  $tx$  when interacting with the white nodes and behaves as if it had proposed  $tx'$  when interacting with the black nodes. Assuming that all nodes start the consensus roughly at the same time and they do not switch the preferred ledger, the protocol does the following:

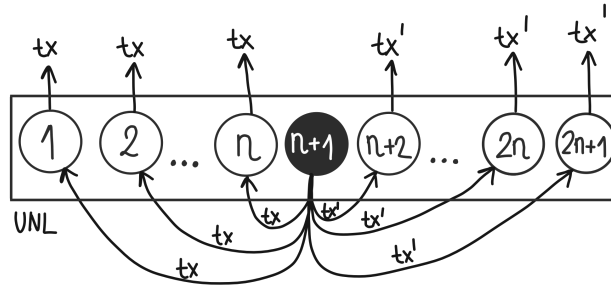
- The Byzantine node 4 submits  $tx$  and  $tx'$  using gossip and causes  $[\text{SUBMIT}, tx]$  to be received by nodes 1, 2, and 3 and  $[\text{SUBMIT}, tx']$  to be received by nodes 5, 6, and 7 from the gossip layer. During the repeated *heartbeat* timer executions in the *open* phase, all correct nodes have the same value of *prevLedger* and send no further messages.
- Suppose at a common execution of the *heartbeat* timer execution (L56) all correct nodes proceed to the *establish* phase and call *closeLedger*. They broadcast the message  $[\text{PROPOSAL}, S]$ , with  $S$  containing  $tx$  or  $tx'$ , respectively (L81). Node 4 sends a PROPOSAL message containing  $tx$  to nodes 1, 2, and 3 and one containing  $tx'$  to nodes 5-7. Furthermore, every correct node executes *createDisputes* with the transaction set *txns* received in each PROPOSAL message, which creates *result.disputes* (L88). For nodes 1, 2, and 3, transaction  $tx'$  is disputed and for nodes 5, 6, and 7, transaction  $tx$  is disputed.
- During *establish* phase, all nodes update their vote for each disputed transaction (L117). Nodes 1, 2, and 3 consider  $tx'$  but do not change their *no* vote on  $tx'$  because only 20% of nodes in their UNL (namely, node 5) vote *yes* on  $tx'$ ; this is less than required *threshold* of 50% or more (L126). The same holds for nodes 5, 6, and 7 with respect to transaction  $tx$ . Hence, *result.txns* remains unchanged and no correct node sends another PROPOSAL message.
- Eventually, function *haveConsensus* returns TRUE for each correct node because the required  $4/5 = 80\%$  of its UNL has issued the same proposal as the node itself (L128). Every correct node moves to the *accepted* phase.
- During *onAccept*, nodes 1, 2, and 3 send a VALIDATION message with ledger  $L = (\text{prevLedger}, \{tx\})$ , whereas nodes 5, 6, and 7 send a VALIDATION message containing  $L' = (\text{prevLedger}, \{tx'\})$  (L137). Node 4 sends a VALIDATION message containing  $tx$  to nodes 1, 2, and 3 and a different one, containing  $tx'$ , to nodes 5, 6, and 7.
- Every correct node subsequently receives five validation messages, from all nodes in its UNL, and finds that 80% among them contain the same ledger (L141). Observe that no node changes its preferred ledger after calling *getPreferred*. This implies that nodes 1, 2, and 3 fully validate  $L$  and execute  $tx$ , whereas nodes 5, 6, and 7 fully validate  $L'$  and execute  $tx'$ . Hence, the agreement condition of consensus is violated.

## 5 Violation of liveness

Here we show how the Ripple consensus protocol may violate liveness even when all nodes have the same UNL and only one node is Byzantine. One can bring the protocol to a state, in which it cannot produce a correct ledger and where it stops making progress.

Consider a system with  $2n$  correct nodes and one single Byzantine node. All nodes are assumed to trust each other, i.e., there is one common UNL containing all  $2n + 1$  nodes. Observe that in this system, the fraction of Byzantines nodes can be made arbitrary small by increasing  $n$ .

As illustrated in Figure 3, node  $n + 1$ , which is Byzantine, exhibits a split-brain behavior and follows the protocol for an input transaction  $tx$  when interacting with nodes  $1, \dots, n$ , and operates with a different input transaction  $tx'$  when interacting with nodes  $n + 2, \dots, 2n +$



■ **Figure 3** Setup in which liveness is violated in the Ripple network. The network consists of  $2n + 1$  nodes with one single UNL and 1 Byzantine (black). The  $n$  first nodes propose transaction  $tx$  while the last  $n$  propose transaction  $tx'$ . The Byzantine proposes transaction  $tx$  to the  $n$  first nodes and transaction  $tx'$  to the last  $n$ .

1. This implies that the first half of the correct nodes, denoted  $1, \dots, n$ , will propose a transaction  $tx$  and the other half, nodes  $n + 2, \dots, 2n + 1$ , will propose transaction  $tx'$ . Similar to the execution shown in Section 4, the nodes start the consensus protocol roughly at the same time and they do not switch the preferred ledger, they proceed like this:

- Byzantine node  $n + 1$  sends two messages,  $[\text{SUBMIT}, tx]$  and  $[\text{SUBMIT}, tx']$ , using the gossip layer and causes  $tx$  to be received by the first  $n$  correct nodes and  $tx'$  to be received by the last  $n$  correct nodes.
- After some time has passed, the correct nodes start to close the ledger and move to the *establish* phase. Every correct node sends a PROPOSAL message, containing only the submitted transaction of which it knows (L81), namely  $tx$  for the first  $n$  correct nodes and  $tx'$  for the last  $n$  correct nodes.
- During *establish* phase, the correct nodes receive the PROPOSAL messages from all nodes (including the Byzantine node) and store them in *currPeerProposals* (L85). Since they all use the same UNL, all obtain the same PROPOSAL messages from the correct nodes.
- Each node creates disputes (L88) and updates them while more PROPOSAL messages arrive. Since the proposed transaction sets differ, each node creates a dispute for  $tx$  and for  $tx'$ .
- While the PROPOSAL messages are being processed, votes are counted in *updateVotes* (L117), using the *yays* and *nays* of each disputed transaction. For a correct node in  $\{1, \dots, n\}$ , notice that the first  $n$  nodes and the Byzantine node vote *no* for  $tx'$  and the last  $n$  nodes vote *yes*. Thus, the fraction of nodes voting *yes* for  $tx'$  is less than required threshold (50%), and so the first  $n$  nodes continue to vote *no* for  $tx'$ . Similarly, nodes  $n + 2$  to  $2n + 1$  never update their vote on  $tx$  and always vote *no* for  $tx$ .
- The *haveConsensus* function called periodically during the *establish* phase checks if at least 80% of the nodes in the UNL agree on the the proposal of the node itself (L128). From the perspective of each one of the first  $n$  correct nodes,  $n$  other nodes agree and  $n$  nodes disagree with its proposal, which contains  $tx$ . That is not enough support for achieving consensus and the function will return FALSE. The same holds from the perspective of the last  $n$  correct nodes, which also continuously return FALSE.
- Finally, the correct nodes will continue trying to update votes and get enough support, but without being able to generate a correct ledger. No correct node proceeds to validating the ledger. In other words, liveness of the protocol is not guaranteed.

## 6 Conclusion

Ripple is one of the oldest public blockchain platforms. For a long time, its native XRP token has been the third-most valuable in terms of its total market capitalization. The Ripple network is implemented as a peer-to-peer network of validator nodes, which should reach consensus even in the presence of faulty or malicious nodes. Its consensus protocol is generally considered to be a Byzantine fault-tolerant protocol, but without global knowledge of all participating nodes and where a node only communicates with other nodes it knows from its UNL. Previous work regarding the Ripple consensus protocol has already brought up some concerns about its liveness and safety. In order to better analyze the protocol, this work has presented an independent, abstract description derived directly from the implementation. Furthermore, this work has identified relatively simple cases, in which the protocol may violate safety and/or liveness and which have devastating effects on the health of the network. Our analysis illustrates the need for very close synchronization, tight interconnection, and fault-free operations among the participating validators in the Ripple network.

---

### References

- 1 Bowen Alpern and Fred B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985. doi:10.1016/0020-0190(85)90056-0.
- 2 Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolic, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: a distributed operating system for permissioned blockchains. In Rui Oliveira, Pascal Felber, and Y. Charlie Hu, editors, *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 30:1–30:15. ACM, 2018. doi:10.1145/3190508.3190538.
- 3 Frederik Armknecht, Ghassan O. Karame, Avikarsha Mandal, Franck Youssef, and Erik Zenner. Ripple: Overview and outlook. In Mauro Conti, Matthias Schunter, and Ioannis G. Askoxylakis, editors, *Trust and Trustworthy Computing - 8th International Conference, TRUST 2015, Heraklion, Greece, August 24-26, 2015, Proceedings*, volume 9229 of *Lecture Notes in Computer Science*, pages 163–180. Springer, 2015. doi:10.1007/978-3-319-22846-4\_10.
- 4 Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. *CoRR*, abs/1807.04938, 2018. arXiv:1807.04938.
- 5 Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011. doi:10.1007/978-3-642-15260-3.
- 6 Christian Cachin and Björn Tackmann. Asymmetric distributed trust. In Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller, editors, *23rd International Conference on Principles of Distributed Systems, OPODIS 2019, December 17-19, 2019, Neuchâtel, Switzerland*, volume 153 of *LIPICs*, pages 7:1–7:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICs.OPODIS.2019.7.
- 7 Christian Cachin and Marko Vukolic. Blockchain consensus protocols in the wild (keynote talk). In Andréa W. Richa, editor, *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, volume 91 of *LIPICs*, pages 1:1–1:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICs.DISC.2017.1.
- 8 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002. doi:10.1145/571637.571640.
- 9 Bernadette Charron-Bost, Fernando Pedone, and André Schiper, editors. *Replication: Theory and Practice*, volume 5959 of *Lecture Notes in Computer Science*. Springer, 2010. doi:10.1007/978-3-642-11294-2.

- 10 Brad Chase and Ethan MacBrough. Analysis of the XRP ledger consensus protocol. *CoRR*, abs/1802.07242, 2018. [arXiv:1802.07242](https://arxiv.org/abs/1802.07242).
- 11 Klitias Christodoulou, Elias Iosif, Antonios Inglezakis, and Marinos Themistocleous. Consensus crash testing: Exploring ripple’s decentralization degree in adversarial environments. *Future Internet*, 12(3):53, 2020. doi:10.3390/fi12030053.
- 12 Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988. doi:10.1145/42282.42283.
- 13 Guy Golan-Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K. Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: A scalable and decentralized trust infrastructure. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, June 24-27, 2019*, pages 568–580. IEEE, 2019. doi:10.1109/DSN.2019.00063.
- 14 Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. In Sape J. Mullender, editor, *Distributed Systems (2nd Ed.)*. ACM Press & Addison-Wesley, New York, 1993.
- 15 Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982. doi:10.1145/357172.357176.
- 16 LibraBFT Team. State machine replication in the Libra blockchain. Technical report, 2020. URL: <https://developers.libra.org/docs/state-machine-replication-paper>.
- 17 Giuliano Losa, Eli Gafni, and David Mazières. Stellar consensus by instantiation. In Jukka Suomela, editor, *33rd International Symposium on Distributed Computing, DISC 2019, October 14-18, 2019, Budapest, Hungary*, volume 146 of *LIPICs*, pages 27:1–27:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICs.DISC.2019.27.
- 18 Adriano Di Luzio, Alessandro Mei, and Julinda Stefa. Consensus robustness and transaction de-anonymization in the ripple currency exchange system. In Kisung Lee and Ling Liu, editors, *37th IEEE International Conference on Distributed Computing Systems, ICDCS 2017, Atlanta, GA, USA, June 5-8, 2017*, pages 140–150. IEEE Computer Society, 2017. doi:10.1109/ICDCS.2017.52.
- 19 Lara Mauri, Stelvio Cimato, and Ernesto Damiani. A formal approach for the analysis of the XRP ledger consensus protocol. In Steven Furnell, Paolo Mori, Edgar R. Weippl, and Olivier Camp, editors, *Proceedings of the 6th International Conference on Information Systems Security and Privacy, ICISPP 2020, Valletta, Malta, February 25-27, 2020*, pages 52–63. SCITEPRESS, 2020. doi:10.5220/0008954200520063.
- 20 Pedro Moreno-Sanchez, Navin Modi, Raghuvir Songhela, Aniket Kate, and Sonia Fahmy. Mind your credit: Assessing the health of the ripple credit network. In Pierre-Antoine Champin, Fabien L. Gandon, Mounia Lalmas, and Panagiotis G. Ipeirotis, editors, *Proceedings of the 2018 World Wide Web Conference on World Wide Web, WWW 2018, Lyon, France, April 23-27, 2018*, pages 329–338. ACM, 2018. doi:10.1145/3178876.3186099.
- 21 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Whitepaper, 2009. URL: <http://bitcoin.org/bitcoin.pdf>.
- 22 Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980. doi:10.1145/322186.322188.
- 23 Ripple Labs. Ripple 1.4.0. <https://github.com/ripple/rippled/releases/tag/1.4.0>.
- 24 Ripple Labs. XRP Ledger Documentation > Concepts > Introduction > XRP Ledger Overview. Available online, <https://xrpl.org/xrp-ledger-overview.html>.
- 25 Ripple Labs. XRP Ledger Documentation > Concepts > Consensus Network > Consensus. Available online, <https://xrpl.org/consensus.html>, 2020.
- 26 Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990. doi:10.1145/98163.98167.
- 27 David Schwartz, Noah Youngs, and Arthur Britto. The Ripple protocol consensus algorithm. Ripple Labs Inc., available online, [https://ripple.com/files/ripple\\_consensus\\_whitepaper.pdf](https://ripple.com/files/ripple_consensus_whitepaper.pdf), 2014.



# Information Theoretic HotStuff

**Ittai Abraham**

VMWare Research, Herzliya, Israel

**Gilad Stern**

The Hebrew University in Jerusalem, Israel

---

## Abstract

---

This work presents Information Theoretic HotStuff (IT-HS), a new optimally resilient protocol for solving Byzantine Agreement in partial synchrony with information theoretic security guarantees. In particular, IT-HS does not depend on any PKI or common setup assumptions and is resilient to computationally unbounded adversaries. IT-HS is based on the Primary-Backup view-based paradigm. In IT-HS, in each view, and in each view change, each party sends only a constant number of words to every other party. This yields an  $O(n^2)$  word and message complexity in each view. In addition, IT-HS requires just  $O(1)$  persistent local storage and  $O(n)$  transient local storage. Finally, like all Primary-Backup view-based protocols in partial synchrony, after the system becomes synchronous, all nonfaulty parties decide on a value in the first view a nonfaulty leader is chosen. Moreover, like PBFT and HotStuff, IT-HS is optimistically responsive: with a nonfaulty leader, parties decide as quickly as the network allows them to do so, without regard for the known upper bound on network delay. Our work improves in multiple dimensions upon the information theoretic version of PBFT presented by Miguel Castro, and can be seen as an information theoretic variant of the HotStuff paradigm.

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

**Keywords and phrases** byzantine agreement, partial synchrony, bounded space

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2020.11

**Related Version** A full version of the paper is available at <https://arxiv.org/abs/2009.12828>.

**Funding** *Gilad Stern*: This work was supported by the HUJI Federmann Cyber Security Research Center in conjunction with the Israel National Cyber Directorate (INCD) in the Prime Minister's Office.

## 1 Introduction

This work assumes the model of Castro and Liskov's PBFT protocol [7, 9, 11]. In particular we deal with the task of Byzantine Agreement in a partially synchronous network. The setting of partial synchrony was proposed by Dwork, Lynch, and Stockmeyer [12] and studied extensively since. In this model, the network starts off as an asynchronous network and at some unknown time becomes synchronous with a known delay  $\Delta$  on message arrival. This time is known as the Global Stabilization Time, or GST in short. This model turns out to be a useful one, managing to capture some of the behaviour of real-world networks. As in PBFT, our goal in this work is to reduce the use of cryptographic tools that require a computationally bounded adversary as much as possible. Much like PBFT, our algorithm is *information theoretically secure*. Formally, as in PBFT [7, 9, 11], our protocol is secure against adversaries that are not computationally bounded under the assumption that there exist authenticated channels that can be made secure against such adversaries. For example, authenticated channels can be obtained via a setup of one time pads or via Quantum key exchange [2].



© Ittai Abraham and Gilad Stern;

licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 11; pp. 11:1–11:16

Leibniz International Proceedings in Informatics



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

There are several good reasons to design protocols in the information theoretic security setting. First, from a theoretical perspective we are interested in minimizing the assumptions. Fewer assumptions often tend to add clarity and conceptual simplicity. Secondly, adding public-key cryptography primitives adds a performance overhead and increases the code-base attack surface, whereas computations in the information-theoretic setting are quick and often amount to simple memory management and counting. Finally, protocols in this setting are more “future-proof”. Such protocols are more resilient to breaking certain cryptographic assumptions and to major technological disruptions in the field.

The PBFT variants that use a PKI and digital signatures can easily use bounded storage at each party (per active slot). One of the challenges of the PBFT protocol when only authenticated channels (no signatures) are used is that obtaining bounded storage is not immediate. Indeed all the peer reviewed papers that we are aware of obtain unbounded solutions [7, 10]. Castro’s thesis [9] does include a bounded storage solution, however to the best of our knowledge this result was not published in a peer reviewed venue, and its complexity does rely on cryptographic hash functions.

## 1.1 Main result

Our main result is *Information Theoretic HotStuff* (IT-HS), a protocol solving the task of Byzantine Agreement in partial synchrony with information theoretic security using bounded storage that sends messages whose maximal size is  $O(1)$  words (both during a view and during a view change). The protocol is resilient to any number of Byzantine parties  $f$  such that  $n > 3f$ , making it optimally resilient. In the protocol, there are several virtual rounds called views, and each one has a leader, called a primary. This is a common paradigm for solving Byzantine agreement, famously used in the Paxos protocol [16] and in later iterations on those ideas such as PBFT [7, 9, 15] and more recent protocols in the Blockchain era [4, 5, 6, 14, 18]. We use a standard measure of storage called a *word* and assume a word can contain enough information to store any command, identifier, or counter. Formally, this means that much like in all previous systems and protocols, our counters, identifiers, and views are bounded (by say 256 bits). In IT-HS, in each view and in each view change, each party sends just a constant number of words and messages to each other party, making the total word and message complexity  $O(n^2)$  in each view and in each view change. As far as we know, this is the best known communication complexity and word complexity for information theoretic protocols of this kind (see table below for comparison). In addition, all parties require  $O(n)$  space throughout the protocol, out of which only  $O(1)$  space needs to be persistent, crash-resistant memory. Clearly at least  $O(1)$  persistent memory is required, because otherwise a decided upon value can be “forgotten” by all parties if they crash and reboot. As far as we know,  $O(n)$  transient space complexity is the best known result. In the shared memory model, a lower bound of  $\Omega(n)$  registers exists [13], suggesting that the total amount of persistent memory in the system is optimal.

In IT-HS, all nonfaulty parties are guaranteed to decide on a value and terminate during the first view a nonfaulty party is chosen as primary after GST, if they haven’t done so earlier. This is the asymptotically optimal convergence for such protocols: For deterministic leader rotation this implies  $O(f)$  rounds after GST. If we assume that parties have access to a randomized leader-election beacon, then this implies  $O(1)$  expected rounds after GST. Furthermore, like PBFT and HotStuff, IT-HS is *optimistically responsive*. If the network delay is actually  $\delta = o(\Delta)$ , all nonfaulty parties terminate in  $O(\delta)$  time instead of in  $O(\Delta)$  time. IT-HS uses an *asymptotically* optimal (constant) number of rounds given a nonfaulty primary and after the network becomes synchronous.



The most relevant related works for IT-HS are the PBFT protocol variants [7, 9, 10, 11] and the HotStuff protocol variants [18]. The following table provides a comparison between them.

	Assumptions	Persistent storage	Maximum size of message (in words)
PBFT (OSDI) [11]	<b>PKI</b>	$\Omega(n)$	$O(n)$
PBFT (TOCS) [10]	Authenticated Channels, <b>Cryptographic Hash</b>	$\Omega(n)$ per view <b>(unbounded)</b>	$\Omega(n)$ per view <b>(unbounded)</b>
PBFT (Thesis) [9]	Authenticated Channels, <b>Cryptographic Hash</b>	$O(1)$	$O(n)$
YAVP (Cachin) [7]	Authenticated Channels, <b>Cryptographic Hash</b>	$\Omega(n)$ per view <b>(unbounded)</b>	$\Omega(n)$ per view <b>(unbounded)</b>
HotStuff (authenticators) [18]	<b>PKI</b>	$O(n)$	$O(n)$
HotStuff (threshold sig) [18]	<b>DKG: Threshold signature setup</b>	$O(1)$	$O(1)$ <b>(threshold sig)</b>
<b>IT-HS</b> (this work)	Authenticated Channels	$O(1)$	$O(1)$

As mentioned earlier, all previous peer-reviewed works in the information theoretic setting require at least  $\Omega(n \cdot v)$  words of storage, where  $v$  is the view number. Since the view number can grow arbitrarily large, the persistent storage requirement is unbounded. The only work we know of that achieves comparable asymptotic performance relies on the relatively strong cryptographic assumption of threshold signatures.

We note that IT-HS does not only use fewer assumptions (does not use any cryptographic hash function), it also obtains the asymptotically optimal  $O(1)$  word bound on the maximal message size. All other protocols require at least  $\Omega(n)$  size messages to be sent during view change by the primary (except for Hotstuff when using a Distributed Key Generation setup and threshold signatures).

Compared to PBFT, our work can be seen as addressing the open problem left in the PBFT journal version (which uses unbounded space and cryptographic hash functions) and is an improvement of the non peer-reviewed PBFT thesis work (which still uses cryptographic hash functions). IT-HS obtains the same  $O(1)$  persistent space, and manages to reduce the maximum message size from  $O(n)$  (in the PBFT view change) to the asymptotically optimal  $O(1)$  maximum message size and requires no cryptographic hash functions.

Relative to HotStuff, our work shows that without any PKI (public key infrastructure) or DKG (distributed key generation) assumptions and without any cryptographic setup ceremony, constant size messages and constant size persistent storage are possible! We do note that IT-HS requires  $O(n^2)$  messages and words per view, while the Hotstuff version with a DKG setup that uses threshold signatures requires just  $O(n)$  messages and words per view. On the other hand, HS-IT requires no cryptographic setup ceremony and no computational assumptions other than pairwise authenticated channels. Like HS-IT, all other protocols that do not use threshold signatures (even those that require a PKI) use  $\Omega(n^2)$  words per view.

## Our contributions

1. Unlike previous solutions which used cryptographic hash functions and required  $O(n)$  sized messages, We provide the first information theoretic primary backup protocol where all messages have size  $O(1)$  and storage is bounded to size  $O(1)$ .
2. We manage to reduce the size of the view change messages to a constant by adapting the HotStuff paradigm without using any cryptographic primitives. We introduce an information theoretic technique for one-transferable signatures to maintain bounded space and adopt the view change protocol accordingly.

3. Without using any cryptographic primitives, we obtain a protocol that requires just a constant amount of persistent storage. We use information theoretic techniques that require storing just the last two events from each message type.

## 1.2 Main Techniques

As the name might suggest, IT-HS is inspired by the Tendermint, Casper, and HotStuff protocols [4, 5, 6, 18] and adapts them to the information theoretic setting. We show how to adapt the *lock and key* mechanism which was suggested in HotStuff [18] and made explicit in [1], to the information theoretic setting while maintaining just  $O(1)$  persistent storage. In a basic *locking* mechanism [4, 12], before nonfaulty parties decide on a value, they set a “lock” that doesn’t allow them to respond to primaries suggesting values from older views. Then, before deciding on a value, nonfaulty parties require a proof that enough parties are locked on the current view. This ensures that if some value is decided upon, there will be a large number of nonfaulty parties that won’t be willing to receive messages from older views, and thus this will remain the only viable value in the system.

The challenge with the locking mechanism is that the adversary can cause nonfaulty locked parties to block nonfaulty primaries, unless the primary waits for all nonfaulty parties to respond. To overcome this, an additional round is added so that a nonfaulty locked party guarantees that there is a sufficient number of nonfaulty parties with a *key*. When a new primary is chosen, it waits for just  $n - f$  parties to send their highest keys, and uses the highest one it receives.

The challenge with using a key is verifying its authenticity. In the cryptographic setting, this is easily done using signatures. In the information theoretic setting, verification is more challenging. One approach is using Bracha’s Broadcast [3] in order to prove that the key received by the primary will also be accepted by the other parties. Since there is no indication of termination in Bracha’s Broadcast, there is a need to maintain an unbounded number of broadcast instances (one for each view). Using such techniques requires an unbounded amount of space.

To overcome this challenge with bounded space, we propose a novel approach of using *one-hop transferable* proofs. If before moving to the next round, a nonfaulty party hears from  $n - f$  parties, then it knows it heard from at least  $f + 1$  nonfaulty parties. This means that once the system becomes synchronous, every party will hear from those  $f + 1$  parties and know that at least one of them is nonfaulty. We use this type of “one-hop transferable proof” twice so we have 3 key messages instead of one, each proving that the next key (or lock) is correct, and that this fact can be proven to other parties, thereby ensuring liveness.

In order to send just a constant number of words, we send just the last two times that the value of the *key* was updated. If the final update to *key* happened after a lock was set, and its value is different than the lock’s value, then the lock is safe to open. Otherwise, if the older of the two updates was after the lock’s view then at least in one of those times it was updated to a value other than the lock’s value, and thus the lock is also safe to open. Using this idea, parties can also prove to a primary that a *key3* suggestion is safe. In this case, the parties either show a later view in which the same value was set for *key2*, or two later views in which the value of *key2* was updated. This proof shows that any previous lock either has the same value as *key3*, or can be opened safely regardless of its value. The idea of storing just two lock values appears in Castro’s Thesis [9], we significantly extend this technique to use our novel *one-hop transferable* information theoretic “signatures” combined with the HotStuff keys-lock approach.

### 1.3 Protocol Overview

Much like all primary backup protocols, each view of IT-HS consists of a constant number of rounds. Each party waits to receive  $n - f$  round  $i$  messages before it sends a round  $i + 1$  message (in some rounds there are additional checks). Much like PBFT, each round involves an all-to-all message sending format. Throughout the protocol, parties may set a lock for a given view and value. This lock indicates that any proposal for a different *view, value* pair should not be accepted without ample proof that another value reached advanced stages in a later view. In order to provide that proof, the parties send a *proof* message that helps convince parties with locks to accept messages about a different value if appropriate.

The rounds of IT-HS for a given view can be partitioned into 4 parts:

1. View Change: parties first send a *request* message, indicating that they started the view. Once parties hear the *request* message sent by the primary, they respond with their current suggestion for a value to propose, as well as the view in which this suggestion originated, and additional data which will help validate all nonfaulty parties' suggestions (proofs). After receiving those suggestions, the primary checks whether each suggestion is valid, and once it sees  $n - f$  valid suggestions, it sends a *propose* message for the one that originated in the most recent view.
2. Propose message round: this is where a party checks a proposal relative to its lock. Each party checks if it's locked on the same value as the one proposed, or convinced to override its lock by  $f + 1$  proof messages. If that is the case, it responds by sending an *echo* message.
3. Key message rounds: this is where a key is created that can be later used to unlock parties. After receiving  $n - f$  *echo* messages with the same value, parties send a *key1* message with that value. After receiving  $n - f$  *key1* messages with the same value they send a *key2* message. After receiving  $n - f$  *key2* messages with the same value they send a *key3* message. We use these three rounds in order to obtain transferable information theoretic signatures on the key message.
4. Lock and commit rounds: After receiving  $n - f$  *key3* messages with the same value they *lock* on it and send a *lock* message. After receiving  $n - f$  *lock* messages with the same value they *commit* and send a *done* message.

Before sending a *key1* message, the local *key1*, *key1\_val* and *prev\_key1* fields are updated. These fields contain the last view in which a *key1* message was sent, its value, and the last view a *key1* message was sent with a different value. Similar updates take place for the other *key* fields and the *lock* fields. The *echo*, *lock* and various *key* messages are tagged with the current view, while the *done* message is a protocol-wide message and isn't related to a specific view. Similarly to the mechanism in Bracha Broadcast [3], after receiving  $f + 1$  *done* messages, the message is echoed, and after receiving  $n - f$  messages it is accepted and the parties decide and terminate. If a party sees that this view takes more than the expected time, it sends an *abort* message for the view. The same  $f + 1$  threshold for echoing the *abort* message and  $n - f$  threshold for moving to the next view are implemented in order to achieve the same properties. In order to avoid buffering *request* and *abort* messages, only the messages with the highest view  $v$  are actually stored and are understood as a *request* or *abort* message for any view up to  $v$ .

## 2 Byzantine Agreement in Partial Synchrony

This section deals with the task of Byzantine Agreement in a partially synchronous system. In this model, there exist  $n$  parties who have local clocks and authenticated point-to-point channels to every other party. The system starts off fully asynchronous: the clocks are not

---

**Algorithm 1** IT-HS.
Code for party  $i$  with input  $x_i$ :

---

```

1:  $lock \leftarrow 0, lock\_val \leftarrow x_i$ 
2:  $key3 \leftarrow 0, key3\_val \leftarrow x_i$ 
3:  $key2 \leftarrow 0, key2\_val \leftarrow x_i, prev\_key2 \leftarrow -1$ 
4:  $key1 \leftarrow 0, key1\_val \leftarrow x_i, prev\_key1 \leftarrow -1$ 
5:  $view \leftarrow 0$ 
6:  $\forall j \in [n] \text{ highest\_request}[j] \leftarrow 0$ 
7: continually run  $check\_progress()$  in the background
8: while true do  $\triangleright$  memory from last  $process\_messages$  and  $view\_change$  calls is freed
9:    $cur\_view \leftarrow view$ 
10:  as long as  $cur\_view = view$ , run
11:    at time  $cur\_time() + 11\Delta$  do
12:      send an  $\langle abort, view \rangle$  message to all parties
13:      ignore messages from other views, other than  $abort$ ,  $done$  and  $request$  messages
14:       $primary \leftarrow (view \bmod n) + 1$ 
15:      continually run  $process\_messages(view)$  in the background
16:       $view\_change(view, primary)$ 

```

---

synchronized, and every message can be delayed any finite amount of time before reaching its recipient. At some point in time, the system becomes fully synchronous: the clocks become synchronized, and every message (including the ones previously sent) arrives in  $\Delta$  time at most, for some commonly known  $\Delta$ . It is important to note that even though it is guaranteed that the system eventually becomes synchronous, the parties do not know when it is going to happen, or even if it has already happened. The point in time in which the system becomes synchronous is called the Global Stabilization Time, or GST in short. In the setting of a Byzantine adversary, the adversary can control up to  $f$  parties, making them arbitrarily deviate from the protocol. In general, throughout this work assume that  $f < \frac{n}{3}$ .

► **Definition 1.** *A Byzantine Agreement protocol in partial synchrony has the following properties:*

- **Termination.** *If all nonfaulty parties participate in the protocol, they all eventually decide on a value and terminate.*
- **Correctness.** *If two nonfaulty parties decide on values  $val, val'$ , then  $val = val'$ .*
- **Validity.** *If all parties are nonfaulty and they all have the same input  $val$ , then every nonfaulty party that decides on a value does so with the value  $val$ .*

We note that if we assume the parties have access to an external validity function, as described in [8], this protocol can be easily adjusted to have external validity. In this setting, the external validity function defines which values are “valid”, and all nonfaulty parties are required to output a valid value. The only adjustment needed is for parties to also check if a value is valid before sending an *echo* message.

The main goal of this section is to show that Algorithm 1 is a Byzantine Agreement protocol in partial synchrony resilient to  $f < \frac{n}{3}$  Byzantine parties. For ease of discussion, a party is said to perform an action “in view  $v$ ” if when it performed the action its local *view* variable equaled  $v$ . In addition, we define the notion of messages “supporting” a key or opening a lock:

---

**Algorithm 2** `view_change(view,primary)`.
Code for party  $i$ :

---

```

1: send  $\langle request, view \rangle$  to all parties  $j \in [n]$ 
2: upon  $highest\_request[primary] = view$ , do
3:   send  $\langle suggest, key3, key3\_val, key2, key2\_val, prev\_key2, view \rangle$  to  $primary$ 
4:  $send\_all\_upon\_join(\langle proof, key1, key1\_val, prev\_key1, view \rangle)$ 
5: if  $primary = i$  then
6:    $suggestions \leftarrow \emptyset$ 
7:    $key2\_proofs \leftarrow \emptyset$ 
8:   upon receiving the first  $\langle suggest, k3, v3, k2, v2, pk2, view \rangle$  message from  $j$ , do
9:     if  $pk2 < k2 < view$  then
10:      add  $(k2, v2, pk2)$  to  $key2\_proofs$ 
11:     if  $k3 = 0$  then
12:       add  $(k3, v3)$  to  $suggestions$ 
13:     else if  $k3 < view$  then
14:       upon  $accept\_key(k3, v3, key2\_proofs) = true$ , do
15:         add  $(k3, v3)$  to  $suggestions$ 
16:   wait until  $|suggestions| \geq n - f$ , then do
17:     let  $(k, v) \in suggestions$  be some tuple such that  $\forall (k', v') \in suggestions \ k' \leq k$ 
18:      $send\_all\_upon\_join(\langle propose, k, v, view \rangle)$ 

```

---

► **Definition 2.** A *suggest* message is said to support the pair  $key3, key3\_val$ , if its  $key2, key2\_val$ , and  $prev\_key2$  fields are ones for which at least one of the conditions in the loop of Algorithm 3 is true.

A *proof* message is said to support opening the pair  $lock, lock\_val$  if its  $key1, key1\_val$ , and  $prev\_key1$  fields are ones for which at least one of the conditions in the loop of Algorithm 7 is true.

Before proving that Algorithm 1 is a Byzantine Agreement protocol in partial synchrony, we prove several lemmas. The lemmas can be classified into two types: safety lemmas and liveness lemmas. The safety lemmas show that if a nonfaulty party decides on some value, no nonfaulty party decides on a different value. This is achieved by the locking mechanism. Roughly speaking, if some nonfaulty party decides on some value, there exist  $f + 1$  nonfaulty parties that are locked on that value and will stop any other value from progressing past the *propose* message. The liveness lemmas show two crucial properties for liveness. First of all, if some nonfaulty party sets  $key3$  to be some value, then there are  $f + 1$  parties that will support that key. This means that if a nonfaulty party hears key suggestions from all nonfaulty parties, it accepts them and picks some key. Secondly, if some nonfaulty primary picks a key to propose, the *suggest* messages it receives guarantee that any nonfaulty party will receive enough supporting *proof* messages. This means that all nonfaulty parties eventually accept the primary's proposal, even if they are locked on some other value. In the following lemmas assume that the number of faulty parties is  $f < \frac{n}{3}$ .

## 2.1 Safety Lemmas

The following lemma and corollary show that a primary cannot equivocate in a given view. More precisely, in a given view all nonfaulty parties send messages that report the same value, other than *echo* messages which might have more than one value. The proofs of the lemma and corollary consist of simple counting arguments and are omitted.

■ **Algorithm 3** `accept_key(key,value,proofs)`.

---

```

1: supporting  $\leftarrow 0$ 
2: for all  $(k, v, pk) \in proofs$  do
3:   if  $key \leq pk$  then
4:     supporting  $\leftarrow supporting + 1$ 
5:   else if  $key \leq k \wedge value = v$  then
6:     supporting  $\leftarrow supporting + 1$ 
7: if  $supporting \geq f + 1$  then
8:   return true
9: else
10:  return false

```

---

■ **Algorithm 4** `send_all_upon_join(message)`.

Code for party  $i$ :

```

1: for all parties  $j \in [n]$  do
2:   upon highest_request[ $j$ ] = view, do
3:     send message to party  $j$ 

```

---

► **Lemma 3.** *If two nonfaulty parties send the messages  $\langle key1, val, v \rangle$  and  $\langle key1, val', v \rangle$ , then  $val = val'$ .*

► **Corollary 4.** *If two nonfaulty parties  $i$  and  $j$  send a  $\langle tag, val, v \rangle$  and  $\langle tag', val', v \rangle$  message such that  $tag, tag' \in \{key1, key2, key3, lock\}$  then  $val = val'$ .*

The following lemma and corollary now show that all *done* messages that nonfaulty parties send have the same value. There are two ways nonfaulty party might send a *done* message: in the end of a view, or after receiving enough *done* messages from other parties. In the first view a nonfaulty party sends a *done* message in line 29, no nonfaulty party sends a *done* message with another value because of the previous non-equivocation claims. Then, once such a *done* message is sent, there are  $f + 1$  nonfaulty parties that are locked on that value, and won't allow any other value to be proposed by a primary. Since all nonfaulty parties send *done* messages with the same value at the end of views, they never receive enough *done* messages with another value for them to echo that *done* message.

► **Lemma 5.** *If two nonfaulty parties send the messages  $\langle done, val \rangle$  and  $\langle done, val' \rangle$  in line 29, then  $val = val'$ .*

► **Corollary 6.** *If two nonfaulty parties send the messages  $\langle done, val \rangle$  and  $\langle done, val' \rangle$ , then  $val = val'$ .*

The proofs of Lemma 5 and Corollary 6 closely follow the above description and are omitted.

## 2.2 Liveness Lemmas

The first two lemmas show that no nonfaulty party gets “stuck” in a view. If some nonfaulty party terminates, then every nonfaulty party eventually terminates as well. In addition, after GST, all nonfaulty parties start participating in consecutive views until terminating.

---

**Algorithm 5** `check_progress()`.
 

---

Code for party  $i$ :

```

1:  $\forall j \in [n]$   $highest\_abort[j] \leftarrow 0$ 
2: upon receiving a  $\langle request, v \rangle$  message from party  $j$ , do
3:   if  $highest\_request[j] < v$  then
4:      $highest\_request[j] \leftarrow v$ 
5: upon receiving a  $\langle done, val \rangle$  message from  $f + 1$  parties with the same  $val$ , do
6:   if no  $done$  message has been previously sent then
7:     send  $\langle done, val \rangle$  to every party  $j \in [n]$ 
8: upon receiving a  $\langle done, val \rangle$  message from  $n - f$  parties with the same  $val$ , do
9:   decide  $val$  and terminate
10: upon receiving an  $\langle abort, v \rangle$  message from party  $j$ , do
11:   if  $highest\_abort[j] < v$  then
12:      $highest\_abort[j] \leftarrow v$ 
13:     let  $u$  be the  $f + 1$ 'th largest value in  $highest\_abort$ 
14:     if  $u > highest\_abort[i]$  then
15:       send  $\langle abort, u \rangle$  to every party  $j \in [n]$ 
16:        $highest\_abort[i] \leftarrow u$ 
17:     let  $w$  be the  $n - f$ 'th largest value in  $highest\_abort$ 
18:     if  $w \geq view$  then
19:        $view \leftarrow w + 1$ 

```

---

► **Lemma 7.** *Observe some nonfaulty party  $i$  that terminates. All nonfaulty parties terminate no later than  $2\Delta$  time after both GST occurs, and  $i$  terminates.*

► **Lemma 8.** *Let  $v$  be the highest view that some nonfaulty party is in at GST. For every view  $v' > v$ , all nonfaulty parties either start view  $v'$ , or terminate in some earlier view.*

*Furthermore, if some nonfaulty party starts view  $v'$  after GST, all nonfaulty parties either terminate or start view  $v'$  no later than  $2\Delta$  time afterwards.*

The proofs are straightforward and are omitted. Eventually, all nonfaulty parties participate in some view with a nonfaulty primary, if they haven't terminated previously. The next lemmas show that once that happens, all nonfaulty parties terminate. First of all, in order for that to happen, a primary needs to receive enough suggestions for a  $key3$  that it will accept. The following lemma shows that every nonfaulty party's  $key3$  field has enough support from nonfaulty parties for the primary to accept the key. Intuitively, since a nonfaulty party set its  $key3$  field to some value, there exist  $f + 1$  nonfaulty parties that sent a  $key2$  message with that value. The lemma shows that those  $f + 1$  nonfaulty parties have  $key2$ ,  $key2\_val$  and  $prev\_key2$  fields that continue to support the key.

► **Lemma 9.** *If some nonfaulty party sets  $key3 = v$ ,  $key3\_val = val$  in view  $v$ , then there exist  $f + 1$  nonfaulty parties whose suggest messages in every view  $v' > v$  support  $key3$  and  $key3\_val$ .*

**Proof.** We will prove by induction that there exist  $f + 1$  nonfaulty parties for whom in every  $v' > v$  either  $prev\_key2 \geq key3$ , or  $key2 \geq key$  and  $key2\_value = val$ . Since those are the fields that nonfaulty parties send in *suggest* messages, that proves the lemma. First, observe view  $v$ . In that view, some nonfaulty party set  $key3 = v$  and  $key3\_val = val$ . This means



---

**Algorithm 6** process\_messages(view).
 

---

Code for party  $i$ :

```

1:  $proofs \leftarrow \emptyset$ 
2: upon receiving the first  $\langle proof, k1, v1, pk1, view \rangle$  message from  $j$ , do
3:   if  $view > k1 > pk1$  then
4:     add  $(k1, v1, pk1)$  to  $proofs$ 
5: upon receiving the first  $\langle propose, key, val, view \rangle$  message from  $primary$ , do
6:   if  $lock = 0 \vee val = lock\_val$  then
7:      $send\_all\_upon\_join(\langle echo, val, view \rangle)$ 
8:   else if  $view > key \geq lock$  then
9:     upon  $open\_lock(proofs) = true$ , do
10:       $send\_all\_upon\_join(\langle echo, val, view \rangle)$ 
11: upon receiving an  $\langle echo, val, view \rangle$  message from  $n - f$  parties with the same  $val$ , do
12:    $send\_all\_upon\_join(\langle key1, val, view \rangle)$ 
13:   if  $key1\_val \neq val$  then
14:      $prev\_key1 \leftarrow key1, key1\_val \leftarrow val$ 
15:    $key1 \leftarrow view$ 
16: upon receiving a  $\langle key1, val, view \rangle$  message from  $n - f$  parties with the same  $val$ , do
17:    $send\_all\_upon\_join(\langle key2, val, view \rangle)$ 
18:   if  $key2\_val \neq val$  then
19:      $prev\_key2 \leftarrow key2, key2\_val \leftarrow val$ 
20:    $key2 \leftarrow view$ 
21: upon receiving a  $\langle key2, val, view \rangle$  message from  $n - f$  parties with the same  $val$ , do
22:    $send\_all\_upon\_join(\langle key3, val, view \rangle)$ 
23:    $key3 \leftarrow view, key3\_val \leftarrow val$ 
24: upon receiving a  $\langle key3, val, view \rangle$  message from  $n - f$  parties with the same  $val$ , do
25:    $send\_all\_upon\_join(\langle lock, val, view \rangle)$  to every party  $j \in [n]$ 
26:    $lock \leftarrow view, lock\_val \leftarrow val$ 
27: upon receiving a  $\langle lock, val, view \rangle$  message from  $n - f$  parties with the same  $val$ , do
28:   if no  $done$  message has been previously sent then
29:     send  $\langle done, val \rangle$  to every party  $j \in [n]$ 

```

---

that it received a  $\langle key2, val, v \rangle$  message from  $n - f$  parties,  $f + 1$  of whom are nonfaulty. In addition to other possible updates, every one of those parties updates  $key2 = view$ , and  $key2\_val = val$  if that isn't true already. Those  $f + 1$  parties prove the claim for view  $v$ .

Now assume the claim holds for every  $v'' < v'$ . Observe party  $j$ , which is one of the  $f + 1$  parties described in the induction claim. If  $j$  doesn't update any of its  $key2$  fields in view  $v'$ , those conditions continue to hold in the end of view  $v'$  and in the beginning of the next view. If  $j$  only updates  $key2$  to be  $v'$ , then if  $prev\_key2 \geq key3$ , it remains that way, and if  $key2 \geq key3$  as well as  $key2\_val = key3\_val$ , after updating  $key2$  to be  $v' > key2 \geq key3$ , it also remains that way. Otherwise  $j$  updates  $prev\_key2 = key2$  too. Note that  $key2 > prev\_key2$  at all times. Therefore, before updating  $prev\_key2$ , regardless of which part of the induction claim holds,  $key2 \geq key3$ . After updating  $prev\_key2$  to be  $key2$ ,  $prev\_key2 \geq key3$ , completing the proof. ◀

The following lemma is used to show that if a nonfaulty primary chose some key, and some nonfaulty party has a lock, it is either the case that the key's value equals the lock's value, or there are enough nonfaulty parties that support opening the lock. Note that the conditions of



---

**Algorithm 7** `open_lock(proofs)`.
 

---

Code for party  $i$ :

```

1: supporting  $\leftarrow 0$ 
2: for all  $(k, v, pk) \in \textit{proofs}$  do
3:   if  $\textit{lock} \leq pk$  then
4:     supporting  $\leftarrow \textit{supporting} + 1$ 
5:   else if  $\textit{lock} \leq k \wedge v \neq \textit{lock\_val}$  then
6:     supporting  $\leftarrow \textit{supporting} + 1$ 
7: if  $\textit{supporting} \geq f + 1$  then
8:   return true
9: else
10:  return false

```

---

the lemma are nearly identical to the conditions the primary checks before accepting a proof as supporting some key. This means that before accepting a key, the primary essentially checks if there is enough support to open any other lock. Similarly to the previous lemma, this lemma shows that if some nonfaulty party sets  $\textit{key2}$  to some value, there are  $f + 1$  parties that sent a  $\textit{key1}$  message with that value. Those  $f + 1$  parties'  $\textit{key1}$ ,  $\textit{key1\_val}$  and  $\textit{prev\_key1}$  fields then continue to support any lock set previously with another value.

► **Lemma 10.** *Let  $\textit{lock} > 0$  be some nonfaulty party's lock and  $\textit{lock\_val}$  be its value. If some nonfaulty party either has  $\textit{prev\_key2} \geq \textit{lock}$  or  $\textit{key2} \geq \textit{lock}$  and  $\textit{key2\_val} \neq \textit{lock\_val}$ , then there exist  $f + 1$  nonfaulty parties whose  $\textit{key1}$ ,  $\textit{key1\_val}$  and  $\textit{prev\_key1}$  fields support opening the lock.*

**Proof.** Let  $i$  be a nonfaulty party such that either  $\textit{prev\_key2} \geq \textit{lock}$  or  $\textit{key2} \geq \textit{lock}$  and  $\textit{key2\_val} \neq \textit{lock\_val}$ . If  $\textit{key2} \geq \textit{lock} > 0$  and  $\textit{key2\_val} \neq \textit{lock\_val}$ ,  $i$  received a  $\langle \textit{key1}, \textit{key1\_val}, \textit{key2} \rangle$  message from  $n - f$  parties in view  $\textit{key2}$ . Out of those  $n - f$  parties, at least  $f + 1$  are nonfaulty. On the other hand, if  $\textit{prev\_key2} \geq \textit{lock} > 0$ , then for some pair of values  $\textit{val}, \textit{val}'$  such that  $\textit{val} \neq \textit{val}'$ ,  $i$  received a  $\langle \textit{key1}, \textit{val}, \textit{prev\_key2} \rangle$  message from  $f + 1$  nonfaulty parties in view  $\textit{prev\_key2}$  and a  $\langle \textit{key1}, \textit{val}', \textit{key2} \rangle$  message from  $f + 1$  nonfaulty parties in view  $\textit{key2} > \textit{prev\_key2} \geq \textit{lock}$ . At least one of the values  $\textit{val}, \textit{val}'$  must not equal  $\textit{lock\_val}$  because  $\textit{val} \neq \textit{val}'$ . In other words, in both cases there exist  $f + 1$  nonfaulty parties that sent a  $\langle \textit{key1}, \textit{val}, v \rangle$  in view  $v$  such that  $\textit{val} \neq \textit{lock\_val}$  and  $v \geq \textit{lock}$ . Let  $I$  be the set of those nonfaulty parties.

We now prove by induction that for every  $v' \geq v$ , all of the parties in  $I$  either have  $\textit{prev\_key1} \geq \textit{lock}$  or  $\textit{key1} \geq \textit{lock}$  and  $\textit{key1\_val} \neq \textit{lock\_val}$ . First, observe view  $v$ . As stated above, in view  $v$  all of the parties in  $I$  sent a  $\langle \textit{key1}, \textit{val}, v \rangle$  and thus set  $\textit{key1} = v \geq \textit{lock}$  and  $\textit{key1\_val} = \textit{val} \neq \textit{lock\_val}$ , if it wasn't already so. Now, assume the claim holds for all views  $v'' < v'$ . Note that the values of  $\textit{key1}$  and  $\textit{prev\_key1}$  only grow throughout the run. This means that if  $\textit{prev\_key1} \geq \textit{lock}$  in the beginning of view  $v'$ , this will also be true at the end of view  $v'$ . On the other hand, if that is not the case, then in the beginning of view  $v'$ ,  $\textit{key1} \geq \textit{lock}$  and  $\textit{key1\_val} \neq \textit{lock\_val}$ . If the value of  $\textit{key1\_val}$  isn't updated in view  $v'$ , then  $\textit{key1}$  can only grow and thus the claim continues to hold. On the other, if the value of  $\textit{key1\_val}$  is updated in view  $v'$ , then for some  $\textit{val}' \neq \textit{val}$  the following updates take place:  $\textit{key1} \leftarrow v', \textit{key1\_val} \leftarrow \textit{val}', \textit{prev\_key1} \leftarrow \textit{key1}$ . By assumption, in the beginning of view  $v'$ ,  $\textit{key1} \geq \textit{lock}$ , and thus after the update  $\textit{prev\_key1} \geq \textit{lock}$ , completing the proof. ◀

This final lemma ties the two previous lemmas together. Once a nonfaulty party is chosen as primary after GST, the primary receives enough keys, and each one of them has enough support to be accepted. Then, after the key is sent, every nonfaulty party either has a lock with the same value, or there is enough support to open its lock. From this point on, the view progresses easily and all nonfaulty parties terminate.

► **Lemma 11.** *Let  $v$  be the first view with a nonfaulty primary that starts after GST<sup>1</sup>. All nonfaulty parties decide on a value and terminate in view  $v$ , if they haven't done so earlier.*

*Furthermore, if all messages between nonfaulty parties are actually delayed only  $\delta$  time until being received, they decide on a value and terminate in  $O(\delta)$  time.*

The proof of the lemma follows naturally from the previous lemmas and is omitted.

### 2.3 Main Theorem

Using the previous lemmas, it is now possible to prove the main theorem:

► **Theorem 12.** *Algorithm 1 is a Byzantine Agreement protocol in partial synchrony resilient to  $f < \frac{n}{3}$  Byzantine parties.*

**Proof.** We prove each property individually.

**Correctness.** Observe two nonfaulty parties  $i, j$  that decide on the values  $val, val'$  respectively.

Party  $i$  first received a  $\langle done, val \rangle$  message from  $n - f$  parties, and  $j$  received a  $\langle done, val' \rangle$  message from  $n - f$  parties. Since  $n - f > f$ ,  $i$  and  $j$  receive at least one of their respective messages from some nonfaulty party. From Corollary 6, all nonfaulty parties that send a *done* message do so with the same value. Therefore,  $val = val'$ .

**Validity.** Assume that all parties are nonfaulty and that they have the same input  $val$ . We will prove by induction that for every view  $v$ , every nonfaulty party has  $key3\_val = val$ . Furthermore, if some nonfaulty party sends a  $\langle key1, val', v \rangle$  message, then  $val' = val$ . First, all parties set  $key3\_val$  to be  $val$  in the beginning of the protocol. Assume the claim holds for every  $v' < v$ . In the beginning of view  $v$ , the primary calls the *view\_change* protocol. Before completing *view\_change*, the primary receives *suggest* messages from  $n - f$  parties with their  $key3\_val$  field. Since all parties are nonfaulty, they all send the  $key3\_val$  they have at that point, and from the induction hypothesis  $key3\_val = val$ . This means that if the primary completes the *view\_change* protocol, it sees that for every  $(key, key\_val) \in suggestions$ ,  $key\_val = val$  and thus if the primary sends a *propose* message it sends the message  $\langle propose, val, key, v \rangle$  to all parties. Now, every nonfaulty party that sends a *key1* message sends the message  $\langle key1, val, v \rangle$ . From Corollary 4, every nonfaulty party that sends a  $\langle key3, val', v \rangle$  message, does so with  $val' = val$ . If a nonfaulty party updates  $key3\_val$  to a new value  $val'$ , it also sends a  $\langle key3, val', v \rangle$  message. However, as shown above the only value sent in such a message is  $val$  so no nonfaulty party updates its  $key3\_val$  field to any other value. Using Corollary 4, every nonfaulty party that sends a *lock* message does so with the value  $val$ . This means that any party that sends a *done* message in line 29, does so with the value  $val$ . Clearly any party that sends a *done* message in line 7 does so with the value  $val$  as well, because it never receives *done* messages with any other value. Finally, this means that every nonfaulty party that decides on a value decides on  $val$ .

<sup>1</sup> More precisely, by “starting after GST”, we mean that the first time some nonfaulty party has  $view \geq v$  is after GST.

**Termination.** Observe the system after GST, and let  $v$  be the highest view that some nonfaulty party is in at that time. From Lemma 8, all nonfaulty parties either terminate or participate in every view  $v' > v$ . Since the primaries are chosen in a round-robin fashion, after no more than  $f + 1$  views, some nonfaulty party starts a view with a nonfaulty primary. From Lemma 11, all nonfaulty parties either terminate in that view or earlier. ◀

## 2.4 Complexity Measures

The main complexity measures of interest are round complexity, word complexity, and space complexity.

**Word complexity.** In IT-HS in every round, every party sends at most  $O(1)$  words to every other party. We assume that a *word* is large enough to contain any counter or identifier. This implies that just  $O(n^2)$  words are sent in each round.

**Round complexity.** As IT-HS is a primary-backup view-based protocol (like Paxos and PBFT), there are no bounds on the number of rounds while the system is still asynchronous. Therefore, we use the standard measure of counting the number of rounds and number of words sent after GST. Furthermore, in order to be useful in the task of agreeing on many values, a desirable property is *optimistic responsiveness*: when the primary is nonfaulty and the network delay is low, all nonfaulty parties complete the protocol at network speed. This desire is captured in the next definition:

▶ **Definition 13** (Optimistic Responsivness). *Assume all messages between nonfaulty parties are actually delivered in  $\delta < \Delta$  time. The protocol is said to be optimistically responsive if all nonfaulty parties complete the protocol in  $O(\delta)$  time after a nonfaulty primary is chosen after GST.*

**Space complexity.** We separate the local space complexity into two types: persistent memory and transient memory. In this setting, parties can crash and be rebooted. Persistent memory is never erased, even in the event of a crash, while transient memory can be erased by a reboot event. IT-HS requires asymptotically optimal  $O(1)$  persistent storage (measured in words) and just  $O(1)$  transient memory per communication channel (so a total of  $O(n)$  transient memory).

After a reboot, nonfaulty parties can ask other parties to send messages that help recover information needed in their transient memory. In this setting we assume that all nonfaulty parties that terminate still reply to messages asking for previously sent information.

▶ **Theorem 14.** *During Algorithm 1, each nonfaulty party sends a constant number of words to each other party in each view and requires  $O(n)$  memory overall, out of which  $O(1)$  is persistent memory. Furthermore, the protocol is optimistically responsive.*

**Proof.** First note that each view consists of one message sent from all parties to the primary, one message sent from the primary to all parties, and a constant number of all-to-all communication rounds. In addition, each message consists of no more than 7 words. Overall, each party only sends a constant number of messages to every party, each with a constant number of words. In each view, every nonfaulty party needs to remember which messages were sent to it by other parties, as well as a constant amount of information about every *suggest* and *proof* message. Since a constant number of words and messages is sent from each

party to every other party, this requires  $O(n)$  memory. Note that once a new view is started, all of the information stored in the previous call to *view\_change* and *process\_messages* is freed. Other than that, every nonfaulty party allocates two arrays of size  $n$ , a constant number of other fields, and needs to remember the first *done* messages received from every other party. This also requires  $O(n)$  memory. Overall, the only fields that need to be stored in persistent memory are the *view*, *lock*, *lock\_val*, and various *key*, *key\_val* and *prev\_key* fields, as well as the messages it sent in the current view, and the last *done*, *request* and *abort* messages it sent. This is a constant number of fields, in addition to a constant number of messages. After being rebooted, a nonfaulty party  $i$  can ask to receive the last *done*, *request*, and *abort* messages sent by all nonfaulty parties to restore the information it lost that doesn't pertain to any specific view, and any message sent in the current view. In addition, it sends a *request* message for its current view. Upon receiving such a message, a nonfaulty party  $j$  replies with the last *done*, *request* and *abort* messages it sent. In addition, if  $j$  is in the view that party  $i$  asked about, it also re-sends the messages it sent in the current view. Note that this is essentially the same as  $i$  receiving messages late and starting its view after being rebooted, and thus all of the properties still hold. The fact that the protocol is optimistically responsive is proven in Lemma 11. ◀

### 3 Multi-Shot Byzantine Agreement and State Machine Replication

This section describes taking a Byzantine Agreement protocol and using it to solve two tasks that are natural extensions of a single shot agreement. Both tasks deal with different formulations for the idea of agreeing on many values, instead of just one.

#### 3.1 State Machine Replication with Stable Leader (a la PBFT)

In the task of State Machine Replication [17], all parties (called replicas) have knowledge of the same state machine. Each party receives a (possibly infinite) series of instructions to perform on the state machine as input. The goal of the parties is to all perform the same actions on the state machine in the same order. More precisely, the parties are actually only interested in the state of the state machine, and aren't required to see all of the intermediary states throughout computation. In order to avoid trivial solutions, if all parties are nonfaulty and they have the same  $s$ 'th instruction as input, then they all execute it as the  $s$ 'th instruction for the state machine. This task can be achieved utilizing any Byzantine Agreement protocol, using ideas from the PBFT protocol.

In addition to the inputs, the protocol is parameterized by a window size  $\alpha$ . All parties participate in  $\alpha$  instances of the Byzantine Agreement protocol, each one tagged with the current decision number. After each decision, every party saves a log of their current decision, and updates the state machine according to the decided upon instruction. Then, after every  $\frac{\alpha}{2}$  decisions, each party saves a "checkpoint" with the current state of the state machine, and deletes the log of the  $\frac{\alpha}{2}$  oldest decisions. Then, before starting the next  $\frac{\alpha}{2}$  decisions, every party sends its current checkpoint and makes sure it receives the same state from  $n - f$  parties using techniques similar to Bracha broadcast. Furthermore, as long as no view fails, the primary isn't replaced. This means that eventually at some point, either there exists a faulty primary that always acts like a nonfaulty primary, or a nonfaulty primary is chosen and is never replaced. Both sending the checkpoints and replacing faulty leaders require more implementation details which can be found in [9].

Using these techniques, all parties can decide on  $O(\alpha)$  instructions at a time, improving the throughput of the algorithm. The communication complexity per view remains similar to the communication complexity of the IT-HS algorithm, but once a nonfaulty primary

is reached after GST, all invocations of the protocol require only one view to terminate. Alternatively, if a nonfaulty primary is never reached after GST, a faulty party acts like a nonfaulty primary indefinitely, which yields the same round complexity. Finally, if we assume that a description of the state machine requires  $O(S)$  space, the protocol now requires  $O(S + \alpha)$  persistent space in order to store the checkpoints and store the  $O(1)$  state for each slot in the window. In addition, the protocol requires  $O(\alpha \cdot n + S)$  transient space in order to store the information about all active calls to IT-HS, the  $\alpha$  decisions in the log, and a description of the current state of the state machine.

### 3.2 Multi-Shot Agreement with Pipelining (a la HotStuff)

In contrast, we can take the approach of HotStuff [18] and solve the task of multi-shot agreement. In this task, party  $i$  has an infinite series of inputs  $x_i^1, x_i^2, \dots$ , and the goal of the parties is to agree on an infinite number of values. Each decision is associated with a slot which is the number  $s \in \mathbb{N}$  of the decision made. Each one of these decisions is required to have the agreement properties, i.e.: eventually all nonfaulty parties decide on a value for slot  $s$ , they all decide on the same value, and if all parties are nonfaulty and have the same input  $val$  for slot  $s$ , the decision for the slot is  $val$ .

A naive implementation for this task is to sequentially call separate instances of IT-HS for every slot  $s \in \mathbb{N}$ , each with the input  $(s, x_i^s)$ . In order to improve the throughput of the protocol, after completing an instance of the IT-HS protocol, the parties can continue with the next view and the next primary in the round-robin. This slight adjustment ensures that after GST,  $n - f$  out of every  $n$  views have a decision made, and if messages between nonfaulty parties are only delayed  $\delta$  time, each one of those views requires only  $O(\delta)$  time to reach a decision. Slight adjustments need to be made in that case so that *abort* messages are sent about views regardless of the slot, so that all parties continue participating in the same views throughout the protocol. In addition, messages about different slots need to be ignored.

In the case of the optimistic assumption that most parties are nonfaulty, a significantly more efficient alternative can be gleaned from the HotStuff protocol. This alternative uses a technique called *pipelining* (or chaining). Roughly speaking, in this technique, all parties start slot  $s$  by appending messages, starting on the second round (round, not view) of slot  $s - 1$ . In the case of HT-IS, the protocol can be changed so that *suggest* messages are sent to all parties, and then each party starts slot  $s$  after receiving  $n - f$  *suggest* messages in slot  $s$ . Note that the exact length of timeouts needs to be slightly adjusted, and the details can be found in [18]. In slot  $s$ , a nonfaulty primary appends its current proposal to the proposal it heard in slot  $s - 1$ . Then, before deciding on a value in slot  $s$ , parties check that the decision values in the previous slots agree with the proposal in slot  $s$ . If they do, then the parties agree on the value in this slot as well. In this protocol, each view lasts for  $11\Delta$  time, so if at some point a primary sees that a proposal from 11 views ago failed, it appends its proposal to the first one that it accepted from a previous view. After GST, if there are  $m + 11$  nonfaulty primaries in a row, then the last  $m$  primaries are guaranteed to complete the protocol, and thus add  $m$  decisions in  $(m + 11)\Delta$  time instead of in  $m \cdot 11\Delta$  time. This means that in the optimistic case that a vast majority of parties are nonfaulty, the throughput of this protocol is greatly improved as compared to the naive implementation. In this protocol the communication complexity per view is still  $O(n^2)$  messages, but a larger number of words. However, note that it is not always the case that if a nonfaulty primary is chosen, its proposal is accepted. To obtain bounded memory requirement one needs to add a checkpointing mechanism, similar to PBFT. As in PBFT, only  $O(n)$  transient space and  $O(1)$  persistent space are required per decision in addition to the log of the decisions.

---

**References**

---

- 1 Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically optimal validated asynchronous byzantine agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, page 337–346, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3293611.3331612.
- 2 Charles H. Bennett, François Bessette, Gilles Brassard, Louis Salvail, and John Smolin. Experimental quantum cryptography. *Journal of Cryptology*, 5(1):3–28, January 1992. doi:10.1007/BF00191318.
- 3 Gabriel Bracha. Asynchronous byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, November 1987. doi:10.1016/0890-5401(87)90054-X.
- 4 Ethan Buchman. Tendermint: Byzantine fault tolerance in the age of blockchains. [http://known-production.s3.amazonaws.com/uploads/attachment/file/1814/Buchman\\_Ethan\\_201606\\_Msater%2Bthesis.pdf](http://known-production.s3.amazonaws.com/uploads/attachment/file/1814/Buchman_Ethan_201606_Msater%2Bthesis.pdf), 2016.
- 5 Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on bft consensus, 2018. arXiv:1807.04938.
- 6 Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget, 2017. arXiv:1710.09437.
- 7 Christian Cachin. Yet another visit to paxos. <https://cachin.com/cc/papers/pax.pdf>, 2010.
- 8 Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '01, page 524–541, Berlin, Heidelberg, 2001. Springer-Verlag.
- 9 Miguel Castro. Practical byzantine fault tolerance. <https://www.microsoft.com/en-us/research/wp-content/uploads/2017/01/thesis-mcastro.pdf>, 2001.
- 10 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, November 2002. doi:10.1145/571637.571640.
- 11 Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.
- 12 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988. doi:10.1145/42282.42283.
- 13 Rati Gelashvili. On the optimal space complexity of consensus for anonymous processes, 2015. arXiv:1506.06817.
- 14 Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K. Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. Sbft: a scalable and decentralized trust infrastructure, 2018. arXiv:1804.01626.
- 15 Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative byzantine fault tolerance. *SIGOPS Oper. Syst. Rev.*, 41(6):45–58, October 2007. doi:10.1145/1323293.1294267.
- 16 Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.
- 17 Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990. doi:10.1145/98163.98167.
- 18 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*, PODC '19, page 347–356, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3293611.3331591.



# Rational Behaviors in Committee-Based Blockchains

**Yackolley Amoussou-Guenou**

Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France  
Sorbonne Université, CNRS, LIP6, F-75005 Paris, France

**Bruno Biais**

HEC Paris, 1 Rue de la Libération, 78350 Jouy-en-Josas, France

**Maria Potop-Butucaru**

Sorbonne Université, CNRS, LIP6, F-75005 Paris, France

**Sara Tucci-Piergiovanni**

Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France

---

## Abstract

We study the rational behaviors of participants in committee-based blockchains. Committee-based blockchains rely on specific blockchain consensus that must be guaranteed in presence of rational participants. We consider a simplified blockchain consensus algorithm based on existing or proposed committee-based blockchains that encapsulate the main actions of the participants: *voting* for a block, and *checking its validity*. Knowing that those actions have costs, and achieving the consensus gives rewards to committee members, we study using game theory how strategic participants behave while trying to maximize their gains. We consider different reward schemes, and found that in each setting, there exist equilibria where blockchain consensus is guaranteed; in some settings however, there can be coordination failures hindering consensus. Moreover, we study equilibria with trembling participants, which is a novelty in the context of committee-based blockchains. Trembling participants are rational that can do unintended actions with a low probability. We found that in presence of trembling participants, there exist equilibria where blockchain consensus is guaranteed; however, when only voters are rewarded, there also exist equilibria where validity can be violated.

**2012 ACM Subject Classification** Computer systems organization → Dependable and fault-tolerant systems and networks; Theory of computation → Solution concepts in game theory

**Keywords and phrases** BFT Consensus, Blockchains, Game Theory

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2020.12

## 1 Introduction

Most cryptocurrencies rely on distributed technology ledgers. Each user of the cryptocurrency may have a local copy of the ledger. The most popular among the distributed ledger technologies is probably blockchain. A blockchain is a growing sequence of blocks, where each block contains transactions and is linked to the previous block by containing the hash of the latter. Modifying information in a block changes its hash, and the subsequent blocks should be changed in consequence. Blockchains then offer many guarantees, such as tamper resistance. The number of blocks since the genesis to the current is called the *height* of the blockchain, and there should ideally be only one block per height. The way blockchain systems are built (in particular how to add blocks) can be roughly separated in two classes: (i) *forkable* blockchains, where for each height, one participant is drawn at random and has the charge to produce a new block; or (ii) *committee-based* blockchains, where for each height, a committee is selected and is in charge of agreeing on which block to append next.

Forkable blockchains are the most famous and the most popular. There are many techniques to build such blockchains. The protocol to add a new block in the most popular



© Yackolley Amoussou-Guenou, Bruno Biais, Maria Potop-Butucaru, and Sara Tucci-Piergiovanni; licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 12; pp. 12:1–12:16

Leibniz International Proceedings in Informatics



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

blockchains (Bitcoin [22], Ethereum [25]) is called *proof-of-work* (PoW), introduced in [13]. In PoW, a participant needs to prove that it worked to have the right to add the next block. More in details, for a participant to be selected to add a block in the blockchain, it has to be the first to resolve a crypto-puzzle: the more computing power, the higher the chances are to win. This gives rise to many problems, first to increase the chance to solve the problem faster, one needs specialized equipment and a lot of computing power. All participants do these computations, but there is only one winner. These non-environment-friendly computations are not useful other than to solve the crypto-puzzle. Another issue with PoW is that although the probability of having multiple winners at the same time is extremely low, it is not impossible. From time to time, there are multiple winners and blocks proposed for the same height; these are called *forks*, and to ensure consistency and avoid double-spending, fork management should be implemented. To try solving these issues, some blockchains propose to replace the PoW with other protocols such as *proof-of-stake*, e.g., Ouroboros [18]. In proof-of-stake, the more stakes a stakeholder has in the blockchain, the higher are its chances to add a block to the chain. This solves the problem of energy consumption, but not the presence of fork; the selection of the leader is somehow still random. Proof-of-stake may also introduce some concentration of power by the richest stakeholders. Other *proof-of-\** proposals have been made, but all suffer from the fork issue, and sometimes many more.

On the other hand, there are committee-based blockchains, e.g., Algorand [16], HotStuff [26], Tendermint [7], etc. They have the purpose of avoiding forks by relying, instead of one participant drawn at random for each block, on a committee that has to agree on the next block to add. The committees run blockchain consensus algorithms. Those algorithms are inspired by well-known algorithmic techniques such as the one from classical consensus [8, 12, 19, 21, 24]. Committee-based blockchains can guarantee the absence of forks. Compared to Bitcoin and proof-of-work blockchains, committee-based blockchains seem slower since they require many messages to be exchanged, and the selection of the committee members is a complex problem.

In both cases, forkable or non-forkable, blockchain systems usually have economical or financial advantages, specifically for block creators. These advantages serve to give an incentive to maintain the blockchain. With advantages given, participants of such systems may try to maximize their profit. Those participants do not necessarily want to harm the system; they often want to stay in the system but gain the most from it. Such participants are called *rational*. To avoid blockchains collapsing due to the presence of rational participants, we must study them, and ensure that the blockchain consensus properties always hold.

**Contributions.** In this work, we analyze the behavior of rational participants in committee-based blockchains. We show the different equilibria that exist given different methods of rewarding the committee members. We analyze if the equilibria do satisfy the consensus specifications or not. In particular, we found that there always exist equilibria that satisfy the blockchain consensus properties, but these equilibria are not unique and coordination failures may occur, leading to liveness issues. Let  $\nu$  be the number of votes required for a block to be considered produced. The different equilibria are summarized in Table 1.

Additionally, we introduce the notion of “trembling hand” which to the best of our knowledge is a novelty in distributed systems. The trembling hand can be viewed as a failure of rational participants. The idea of trembling hand and acknowledging errors has been studied in different fields, such as in economics (e.g., [11]), in networks (e.g., [10]), etc. With low probability, the player can tremble and do an unintended action. We conduct the same equilibrium analysis and found that there exist equilibria satisfying the consensus properties.



■ **Table 1** Summary of the Equilibria with Rational Players.

	Reward All	Reward Only Senders
$\nu = 1$	Proposition 5 In equilibrium, exactly one message is sent: <b>Consensus</b>	Proposition 1 In equilibrium, All players send a message: <b>Consensus; but inefficient: too costly</b>
$\nu > 1$	Proposition 7 In equilibrium, either: - No message is sent: <b>No Termination: No block, coordination failure, or</b> - Exactly $\nu$ messages are sent. <b>Consensus</b>	Proposition 3 In equilibrium, either: - No message is sent: <b>No Termination: No block, coordination failure, or</b> - All players send a message: <b>Consensus; but inefficient: too costly</b>

■ **Table 2** Summary of the Equilibria with “Trembling” Players.

	Reward All	Reward Only Senders
$\nu = 1$	Proposition 13 In the equilibrium, one message sent if valid: <b>Consensus</b>	Proposition 9 In the equilibrium, either - $n$ messages always sent: <b>Validity not guaranteed</b> - $n$ messages sent only if valid: <b>Consensus</b>
$\nu > 1$	Proposition 15 In equilibrium, either: - No message is sent: <b>No Termination</b> - $\nu - 1$ messages always sent + 1 if valid: <b>Consensus</b>	Proposition 11 In equilibrium, either: - No message is sent: <b>No Termination</b> - (if $\nu < n$ ) $n$ messages always sent: <b>Validity not guaranteed</b> - $\nu - 1$ messages always sent + $(n - \nu + 1)$ if valid: <b>Consensus</b>

However, there also exist equilibria inducing liveness or safety issues because the consensus properties cannot be guaranteed. Equilibria with trembling participants are summarized in Table 2. In all cases, we found that equilibria, when all committee members are rewarded, are efficient in terms of the number of messages.

**Related work.** Many analyses have been made on strategic behaviors in blockchains. However, they mainly focus on forkable systems (*e.g.*, [6, 14]). To the best of our knowledge, very few works have been dedicated to analyze or discuss the rational behaviors among participants in committee-based blockchains. Some exceptions have to be noted.

The work of Abraham *et al.* in [2] is probably the first to consider strategic behaviors in committee-based blockchains. They introduced interesting incentive mechanisms, but did not provide a formal framework for their analysis, nor did they consider the cost of the actions.

Recently, Fooladgar *et al.* show in [15] that the proposed reward distribution in Algorand does not lead to an equilibrium. Interestingly, as in our paper, [15] considers the cost of actions of the players; but as opposed to us, among other things, players have basically one action, either following the protocol or not, so it either incurring all costs or no cost at all. In our work, we refine the approaches; We consider that multiple actions are available to the players, and that they just pay the costs of the actions they did, and not all of them.

In [4], we provide a framework for the analysis of strategic behaviors in the presence of rational players can either exhibit strategic or adversarial behaviors for committee-based blockchains. We however only considered one reward mechanism and did not study trembling hand effects. In this work, we extend the model in [4]; we consider systems with participants that behave strategically and can exhibit trembling hand effects. Additionally, in this work, we study the behavior of the participants under different reward schemes, as opposed to [4].

Previous works studying rational behavior in consensus algorithms (such as [1, 17, 20]) did not take into consideration the rewards given when a decision is reached, nor the cost of participants’ actions. They usually proposed incentive-compatible protocols. Blockchains highlighting the costs and rewards, we take them into account in our analysis.

## 2 Model

### 2.1 System Model

We consider a system composed of a finite and ordered set  $\Pi$  of  $n$  players, called *committee*, of synchronous sequential players denoted by their index, namely  $\Pi = \{1, \dots, n\}$ .

**Communication.** The players communicate by sending and receiving messages through a *synchronous network*. We assume that the players proceed in rounds. A *round* consists of three sequential phases, in order: the send, the delivery and the compute phase. Since we consider synchronous communication, there is a known upper bound on the message transfer delay. Such upper bound is used by the players to set the duration of their rounds, in particular, the duration of the delivery phase is such that for all players, all messages sent at the beginning of the round are received before the end of the delivery phase. At the end of a round, a player exits from the current round and starts the next one. We assume the existence of a *reliable broadcast* primitive. A broadcast is reliable if the following conditions hold: (i) *safety*: every message delivered by a player has been previously sent by a source, and (ii) *liveness*: every player eventually delivers every message sent by a source. Messages are created with a digital signature, and we assume that digital signatures cannot be forged. When a player  $i$  delivers a message, it knows the player  $j$  that created the message.

**Players Behavior.** We consider that players are *rational*. Rational players are self-interested and their objective is to maximize their expected gain. They will deviate from a prescribed protocol if and only if doing so increases their expected gain. They differ from honest players who always follow the prescribed protocol.

We also consider trembling players. With low probability, an external function can return an unexpected value. They do not want such value, but are not in control of that, and are not aware when the returning value is “normal” or not. They only know the probability of such an event happening. A trembling player is also a rational player.

### 2.2 Consensus in Presence of Rational Players

A blockchain is a growing sequence of blocks. The number of blocks since the genesis to the current is called the height of the blockchain. In committee-based blockchains, for each height, a committee is selected and is in charge of agreeing on which block to append next.

As proposed by many articles (*e.g.*, [3], [5], [9], [16], [26], ...), committee-based blockchains can be developed using consensus algorithms. In particular, at each height, the protocol used by the corresponding committee must implement the consensus. In the section, we adapt the definition of consensus properties to take into account the presence of rational players.

We say that a protocol is a consensus algorithm in presence of rational players if the following properties hold:

- *Termination*: every rational player decides on a value (a block);
- *Agreement*: if two rational players decide respectively on values  $B$  and  $B'$ , then  $B = B'$ ;
- *Validity*: a decided value by any rational player is valid; w.r.t a predefined predicate.

**Problem.** We study the behavior of rational players in a consensus protocol. The goal is to know whether consensus is guaranteed in committee-based blockchains in the presence of rational players. For the study, we use the notion of *Nash equilibrium*, which is intuitively a “stable” situation where no player has an incentive to unilaterally deviate.

The question we answer is: *What are the different Nash equilibria and do they satisfy the consensus properties?* It is important to note that we do not propose a protocol such that all rational behave as honest, but rather study the behavior of rational players in a blockchain consensus algorithm under different reward mechanisms.

### 2.3 Protocol Studied

In committee-based blockchains, for each height, there is a committee supposed to reach a consensus on the block to append. The agreement procedure can be seen as a vote in potentially multiple sequential rounds. Focusing on one height, the consensus procedure is as follows. For each round:

- A proposer is selected for the current round. The proposer of the round proposes a block (the proposal) and send it to the rest of the committee members.
- Once a player receives the proposal, it should check its validity and vote (by sending a message) for the block only if it is valid; otherwise, it should not vote if invalid.

At the end of the round, all committee members collect the vote messages and count them. Let  $\nu$  be the number of votes required for a block to be considered produced (the decision of the consensus). If the proposal receives votes for at least  $\nu$  different committee members, then the block is consider *produced*; otherwise the next round starts with a new proposer, proposing a new block and the procedure restarts until a decision is made. When a player considers a block produced (*i.e.*, collects  $\nu$  votes for the block), due to the communication model we consider, all players will also consider the block as produced since they have the same set of messages at the end of any round.

As explained above, these two phases encapsulate the main and important ideas of consensus protocol for committee-based blockchains. Moreover, Chan and Shi in [9], extended this two phases approach (Propose and Vote) to present multiple algorithms for different communication and failures models; pointing out the importance and sufficiency of these phases in consensus algorithms for blockchains.

In the following, we describe the actions rational players have. We present it as a protocol shown in Algorithm 1. Definition of the game and actions is done in the next section. We consider the choice of (i) checking or not the validity of a block and (ii) sending or not the vote for a proposed block. We consider that the actions of checking the validity of the block and of sending the message (of type vote) are costly.

**Protocol of Rational Players.** Rational players have some freedom at executing the prescribed protocol. We represent their possible actions in Algorithm 1, where specific variables have been introduced; namely,

- $action^{check} \in \{\mathbf{false}, \mathbf{true}\}$ , if the player decides to check the validity of the proposal or not; and
- $action^{send} \in \{\mathbf{false}, \mathbf{true}\}$ , if the player decides to vote for the proposal or not (depending on the validity information the player has about the proposal).

$isProposer(t, h)$  returns the identifier of the proposer for the current round (line 10).

All players sets their actions locally, in more details, player  $i$  sets its action variable  $action^{check}$  (resp.  $action^{send}$ ) by calling the dedicated function  $\sigma_i^{check}$  (resp.  $\sigma_i^{send}$ ) representing its strategy.

The strategy  $\sigma_i^{check}$  determines if  $i$ , the receiving player, chooses to check the validity of the proposal or not, which is a costly action. If the player chooses to check the validity (line 17), it will also update the knowledge it has about the validity of the proposal and it will pay a cost  $c_{check}$ . If otherwise, the player keeps not knowing if the proposal is valid or

■ **Algorithm 1** Pseudo-code for a given height  $h$  modeling the rational player  $i$ 's behavior.

---

```

1: Initialization:
2:    $vote := nil$ 
3:    $t := 0$  /* Current round number */
4:    $decidedValue := nil$ 
5:    $action^{check} := nil$ 
6:    $action^{send} := nil$ 
7:    $validValue[] := \{\perp, \perp, \dots, \perp\}$  /*  $validValue[r] \in \{\perp, false, true\}$  */

8: Round PROPOSE( $t$ ):
9:   Send phase:
10:  if  $i == isProposer(t, h)$  then
11:     $proposal \leftarrow createValidValue(h)$ 
12:    broadcast  $\langle PROPOSE, h, t, proposal \rangle$ 
13:  Delivery phase:
14:  delivery  $\langle PROPOSE, h, t, v \rangle$  from  $proposer(h, t)$ 
15:  Compute phase:
16:   $action^{check} \leftarrow \sigma_i^{check}()$  /*  $\sigma_i^{check}() \in \{false, true\}$  sets the action of checking or not the validity of the proposal */
17:  if  $action^{check} == true$  then
18:     $validValue[r] \leftarrow isValid(v)$  /* The execution of  $isValid(v)$  has a cost  $c_{check}$  */
19:   $action^{send} \leftarrow \sigma_i^{send}(validValue)$  /*  $\sigma_i^{send} : \{\perp, false, true\} \rightarrow \{false, true\}$  sets the action of sending the vote or not */
20:  if  $action^{send} == true$  then
21:     $vote \leftarrow v$  /* The player decides to send the vote, the proposal might be invalid */

22: Round VOTE( $t$ ):
23:  Send phase:
24:  if  $vote \neq nil$  then
25:    broadcast  $\langle VOTE_i, h, t, vote \rangle$  /* The execution of the broadcast has a cost  $c_{send}$  */
26:  Delivery phase:
27:  delivery  $\langle VOTE, h, t, v \rangle$  /* The player collects all the votes for the current height and round */
28:  Compute phase:
29:  if  $|\langle VOTE, h, t, v \rangle| \geq \nu \wedge decidedValue = nil \wedge vote \neq nil \wedge vote = v$  then
30:     $decidedValue = v$ ; exit
31:  else
32:     $vote \leftarrow nil$ 
33:     $t \leftarrow t + 1$ 

```

---

not ( $validValue[t]$  remains at  $\perp$ ). Note that this value remains at  $\perp$  even if the player is the proposer. This is because we assumed, without loss of generality, that checking validity has a cost and that the only way of checking validity is by executing the  $isValid(v)$  function.

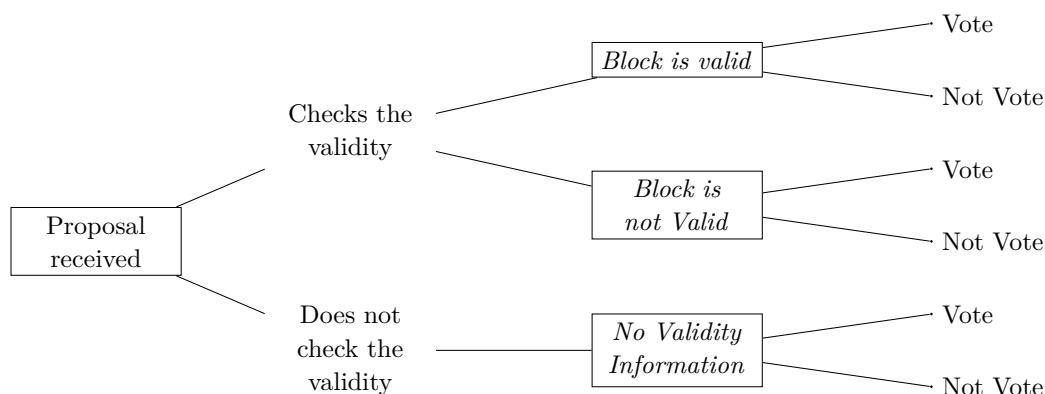
Note that the strategy  $\sigma_i^{send}$  depends on the knowledge the player has about the validity of the proposal. The strategy determines if the player chooses to send its vote for the proposal or not (line 19 - 25). If the player chooses to vote for the proposal, it will pay a cost  $c_{send}$ .

Let us note that a rational player that did not check the validity of the block could consider as decision of the committee an invalid value if it collects more than  $\nu$  votes for an invalid proposal. We also note that in the model model considered, the Agreement property always holds, since, at the end of each round, all players have the same set of messages delivered.

Note that the creation of proposal (line 11 of Algorithm 1) will be subject to the trembling hand effect in Section 5.

## 2.4 Game

**Action space.** At each round  $t$ , when a player receives the proposal, it decides whether to check the block's validity or not (at cost  $c_{check}$ ), and then given the validity information, it decides whether to send a vote message (at cost  $c_{send}$ ) or not.



■ **Figure 1** Decision Tree of one Player after Reception of the Proposal.

**Information sets.** At the beginning of each round  $t > 1$ , the information set of the player,  $\eta_i^t$ , includes the observation of the round number  $t$ , as well as the observation of what happened in previous rounds, namely (i) whether the player decided to check validity, and in that case, it knows the validity of the block, (ii) how many messages were sent, and (iii) whether a block was produced or not.

Then, in each round  $t > 1$ , the player decides whether to check the validity of the current block. At this point, denoting by  $b_t$  the block proposed at round  $t$ , when the player does not decide to check validity  $\text{isValid}(b_t)$  is the null information set, while if the player decides to check,  $\text{isValid}(b_t)$  is equal to 1 if the block is valid and 0 otherwise. Therefore, at this stage, the player information set becomes  $H_i^t = \eta_i^t \cup \text{isValid}(b_t)$ , which is  $\eta_i^t$  augmented with the validity information player  $i$  has about  $b_t$ , the proposed block.

**Strategies.** At each round  $t \geq 1$ , the strategy of player  $i$  is a mapping from its information set into its actions. At the point at which the player can decide to check block validity, its strategy is given by  $\sigma_i^{\text{check}}(\eta_i^t)$ . Finally, after making that decision, the player must decide whether to vote or not, and that decision is given by  $\sigma_i^{\text{send}}(H_i^t)$ . The decision tree of a player is depicted in Figure 1. We note that when the player does not check the validity of the proposal, it does not know if the block is valid or not.

We denote by  $\sigma = (\sigma_1, \dots, \sigma_n)$  the *strategy profile* where  $\forall i \in \{1, \dots, n\}$ , player  $i$  use strategy  $\sigma_i$ , where  $\sigma_i(H_i^t)$  is the pair  $(\sigma_i^{\text{check}}(\eta_i^t), \sigma_i^{\text{send}}(H_i^t))$ .

**Rewards and Costs for the Players.** We study the cases in which:

1. when a block is produced, only the committee members which voted are rewarded (and receive  $R$ ); or
  2. whenever a block is produced, all committee members are rewarded (and receive  $R$ ).
- We will explicitly state the case we are studying.

We also assume that when an invalid block is produced, all players incur a cost  $\kappa^1$ . Note that  $\kappa$  is not incurred when no block is produced. We assume that the reward  $R$ , is larger than the cost  $c_{\text{check}}$  of checking validity, which is larger than the cost  $c_{\text{send}}$  of sending a vote message. Lastly, the reward obtained is smaller than the cost  $\kappa$  of producing an invalid block. That is,  $\kappa > R > c_{\text{check}} > c_{\text{send}} > 0$ .

<sup>1</sup> Such a cost corresponds, for example, to the loss of confidence in the blockchain system caused by the invalid block produced, hence hurting the whole ecosystem.

**Objective of Rational Players.** Let  $T$  be the round at which the game stops. If a block is produced at round  $t \leq n$ , then  $T = t$ . Otherwise, if no block is produced,  $T = n + 1$ . In the latter case, the *termination* property is not satisfied. In our analyses, we focus on what happens during the first round; in particular, when  $T \geq 2$ , we say that termination is not satisfied at round 1.

As explained above, we study two types of rewards. The analyses are done independently. In each setting, all rational players have the same gain function detailed in the following when we focus only on the first round.

1. **Reward Only Sender:** When the reward is given only to players that vote for the produced block, for the first round, the expected gain of rational player  $i$  is:

$$U_i(\sigma) = E \left[ \begin{array}{l} R * \mathbb{1}_{(\sigma_i^{\text{send}}(H_i^1) \wedge \text{block produced at round 1})} - c_{\text{send}} * \mathbb{1}_{\sigma_i^{\text{send}}(H_i^1)} \\ - c_{\text{check}} * \mathbb{1}_{\sigma_i^{\text{check}}(\eta_i^1)} - \kappa * \mathbb{1}_{(\text{invalid block produced at round 1})} \end{array} \right], \quad (1)$$

where  $\mathbb{1}_{(\cdot)}$  denotes the indicator function, taking the value 1 if its argument is true, and 0 if it is false.

2. **Reward All:** When the reward is given to the whole committee once a block is produced, for the first round, the expected gain of rational player  $i$  is:

$$U_i(\sigma) = E \left[ \begin{array}{l} R * \mathbb{1}_{(\text{block produced at round 1})} - c_{\text{send}} * \mathbb{1}_{\sigma_i^{\text{send}}(H_i^1)} \\ - c_{\text{check}} * \mathbb{1}_{\sigma_i^{\text{check}}(\eta_i^1)} - \kappa * \mathbb{1}_{(\text{invalid block produced at round 1})} \end{array} \right]. \quad (2)$$

**Equilibrium concept.** We consider the players are playing Nash equilibria, and we focus only on their behavior during the first round.

Let  $\sigma = (\sigma_1, \dots, \sigma_n)$  be a strategy profile, where  $\sigma_i$  is the strategy of player  $i$ . We write  $(\sigma_{-i}, \sigma'_i)$  to represent the strategy profile  $(\sigma_1, \dots, \sigma_{i-1}, \sigma'_i, \sigma_{i+1}, \dots, \sigma_n)$  where player  $i$  deviates, and the others continue playing their strategy. A strategy profile is a *pure Nash equilibrium* [23] if no player can increase its gain by unilaterally deviating. Formally,  $\sigma$  is a pure Nash equilibrium if and only if  $\forall i \in \{1, \dots, n\}$ , and  $\forall \sigma'_i$  a strategy for  $i$ ,  $U_i(\sigma) \geq U_i((\sigma_{-i}, \sigma'_i))$ . We simply use *Nash equilibrium* instead of pure Nash equilibrium.

The following sections present our results.

In Sections 3 & 4, we do not have trembling hand effects, therefore, we cannot have invalid blocks since the proposal should be valid (line 11 of Algorithm 1). Focusing on liveness issues, we study whether players vote or not in equilibria.

In Section 5, trembling effects are considered, and the proposal may be invalid. Therefore, for safety reasons, players may check the proposal's validity before voting or not.

### 3 Reward Only Committee Members that Vote

In this section, we consider that only committee members that voted for a produced block are rewarded. Equation 1 describes the gain of each rational player.

We study the different equilibria with respect to the value of  $\nu$ , the minimum number of votes required to consider a block as produced.

First, we analyze the case where 1 vote for a proposed block is sufficient to considered it as produced, *i.e.*,  $\nu = 1$ .

► **Proposition 1.** *In one round, with only rational players in the committee, if  $\nu = 1$ , and when only players that vote for the produced block are rewarded, there is only one Nash equilibrium. In the unique equilibrium, all players vote for the proposed block.*

In this equilibrium, all players vote, and the block is produced. No player has an incentive to deviate and not send, such deviation will mean for the player that it will not be rewarded while the block is produced.

► **Remark 2.** Note that in the Nash equilibrium of Proposition 1, the consensus properties are satisfied, in particular, there is always a block produced at the end of the first round.

We now consider the situation where strictly more than one vote is needed to consider a block as produced, *i.e.*,  $\nu \in \{2, \dots, n\}$ .

► **Proposition 3.** *In one round, with only rational players in the committee, if  $\nu > 1$ , and when only players that vote for the produced block are rewarded, there are two Nash equilibria; either (i) all players vote, or (ii) no player votes.*

In the first equilibrium, if a rational player anticipates that no players will vote, its only vote will not make the proposal produced, since  $\nu > 1$ , therefore, the player prefers not voting. In the second type of equilibrium, if a player anticipates that all other players are voting, it prefers voting as well; otherwise, if the player does not send, it will not have a reward.

► **Remark 4.** There are two Nash equilibria in Proposition 3. In the equilibrium where no player votes, Termination is not guarantee at round 1. In the second equilibrium where there are  $n$  votes, the consensus properties are satisfied in the first round.

#### 4 Reward All Committee Members

In this section, we consider that all committee members are rewarded once a block is produced. Equation 2 describes the gain of each rational player.

We study the different equilibria with respect to the value of  $\nu$ , the minimum number of votes required to consider a block as produced.

First, we analyze the case where 1 vote for a proposed block is sufficient to considered it as produced, *i.e.*,  $\nu = 1$ .

► **Proposition 5.** *In one round, with only rational players in the committee, if  $\nu = 1$ , and when all players are rewarded once a block is produced, in the Nash equilibria, exactly one player votes, and the others do nothing.*

If the player supposed to vote does not vote, no block is produced, and hence it does not have any reward. Therefore, it prefers voting, since a block is always produced in equilibrium, if a player not supposed to send deviates and votes, it will pay the cost of sending for nothing since it will be rewarded even without voting.

► **Remark 6.** Note that there exists at most  $n$  equilibria corresponding to Proposition 5. In all the equilibria corresponding to Proposition 5, the consensus properties are satisfied.

We now consider the situation where strictly more than one vote is needed to consider a block as produced, *i.e.*,  $\nu \in \{2, \dots, n\}$ .

► **Proposition 7.** *In one round, with only rational players in the committee, if  $\nu > 1$ , and when all players are rewarded once a block is produced, in the Nash equilibria, either (i) exactly  $\nu$  players vote, or (ii) no player votes.*

If a rational player anticipates that no players will vote, since  $\nu > 1$ , its only vote will not make the proposal produced, therefore, it is better off not voting. In the other type of equilibrium, exactly  $\nu$  players vote; if a player supposed to send does not vote, the block is



## 12:10 Rational Behaviors in Committee-Based Blockchains

not produced and the deviating player is not rewarded any more; if a player not supposed to send deviates (by voting) it will incur a cost of sending, when it will be rewarded in any case, so it prefers not to vote.

- Remark 8. There are two types of Nash equilibria in Proposition 7.
- The equilibrium where no player votes does not guarantee *Termination* at round 1.
- In the second type of equilibrium in this setting, there are exactly  $\nu$  messages sent. There can be at most  $\binom{n}{\nu} + 1$  equilibria corresponding to that setting<sup>2</sup>. In each of them, the consensus properties are satisfied.

A summary of the different equilibria in Sections 3 & 4 can be found in Table 1. When only 1 vote is required to consider a proposal as produced, in all equilibria, blocks are always produced. When we require strictly more than 1 vote to consider a block as produced, although there are equilibria where the consensus is guaranteed, there is also an equilibrium where no player votes, anticipating that the others will not vote as well: a coordination failure, leading to a violation of the Termination. This happens in the two reward mechanisms: reward all committee members, or reward only the members that voted. However, in the equilibria where all committee members are rewarded, less messages are sent, making it a more efficient (and less costly) mechanism with respect to the number of messages.

### 5 Trembling Players at Proposal

Now, we assume that there is some negligible probability  $p$  for the `createValidValue` function (line 11 of Algorithm 1) to return an invalid proposal, and all players are aware of the trembling effect. When proposing a value there is a probability that the hand of the player trembles and proposes an invalid block instead of a valid block; *i.e.*, in some sense, we take into account the possibility of making a mistake for the proposal.

Note that now, checking the validity of a block may be important, there is a risk of producing an invalid block, violating the validity property of the consensus. To ensure that the reward covers the costs of checking and voting, in this setting we assume that  $(1 - p)(R - c_{\text{send}}) - c_{\text{check}} > 0$ . We also note that it is better for the player to vote (resp. not vote) without checking than checking and voting (resp. not voting) irrespective to the block validity; that would mean incurring a cost  $-c_{\text{check}}$  for nothing. It is also not in their best interest to check the validity of the proposal and vote if the proposal is invalid, that would mean increasing the chances of producing an invalid block and incurring a cost  $-\kappa$ . In the analyses, we then consider only the three relevant strategies: a rational player can (i) vote without checking proposal validity, (ii) not vote nor check proposal validity, and (iii) check the proposal validity and vote only if the proposal is valid.

In the following, we make the same analyses as in Sections 3 & 4, *i.e.*, we analyze the behavior of rational players when only voters are rewarded; and their behaviors when all committee members are rewarded.

#### 5.1 Reward Only Committee Members that Vote

In this subsection, we consider that only committee members that voted for a produced block are rewarded. Equation 1 describes the gain of each rational player.

---

<sup>2</sup>  $\binom{n}{\nu} = C_n^\nu$  is the number of combinations for choosing  $\nu$  out of  $n$  elements.



We study the different equilibria with respect to the value of  $\nu$ , the minimum number of votes required to consider a block as produced.

First, we analyze the case where 1 vote for a proposed block is sufficient to consider it as produced, *i.e.*,  $\nu = 1$ .

► **Proposition 9.** *In one round, with only rational players in the committee, if  $\nu = 1$ , when only players that vote for the produced block are rewarded, and if there is a probability  $p$  that the proposer proposes an invalid block, there are two Nash equilibria. In equilibrium, either (i) if  $\kappa \geq R - c_{\text{send}} + c_{\text{check}}/p$ , all players check the validity of the proposal and vote only if it is valid; or (ii) all players vote for the proposal without checking the validity of the proposal.*

As in Proposition 1, one can note that in equilibrium, all players do (try to) vote.

► **Remark 10.** There are two Nash equilibria in Proposition 9. In the equilibrium where all players check and vote, if the proposal is invalid, there is no Termination at the first round, however Validity is always ensured. While in the second equilibrium where no player checks but votes, Termination is always guaranteed at the end of the first round, even if the proposal is invalid, which violates the Validity.

We now consider the situation where strictly more than one vote is needed to consider a block as produced, *i.e.*,  $\nu \in \{2, \dots, n\}$ .

► **Proposition 11.** *In one round, with only rational players in the committee, if  $\nu > 1$ , when only players that vote for the produced block are rewarded, and if there is a probability  $p$  that the proposer proposes an invalid block, there are three Nash equilibria. Either (i) no player votes nor checks the proposal validity; or (ii) if  $\nu < n$ , all players vote for the proposal without checking the validity of the proposal; or (iii) if  $\kappa \geq R - c_{\text{send}} + c_{\text{check}}/p$ ,  $n - \nu + 1$  players check the validity of the proposal and vote only if it is valid, and the  $\nu - 1$  remaining players only vote without checking the validity of the proposal.*

**Proof of Proposition 11.** We prove that the strategy profiles described in the proposition are Nash equilibria.

- First, we prove that the strategy profile where no player votes is a Nash equilibrium. The gain at equilibrium of any player is 0. If one player deviates and votes, there is only 1 vote and the block is not produced since  $\nu > 1$ , the gain at deviation is  $-c_{\text{send}} < 0$ . If the player deviates by checking block validity, it will pay the cost  $-c_{\text{check}} - (1 - p)c_{\text{send}} < 0$ . The strategy profile is indeed a Nash equilibrium.
- We now prove that the strategy profile where all players vote without checking the proposal validity is a Nash equilibrium. Let  $\nu < n$ , the gain at equilibrium of any player is  $R - c_{\text{send}} - p\kappa$ . Even if one player deviates, the block will be produced in any case (since  $\nu < n$ ) no matter its validity. If a player deviates by checking validity and voting if the proposal is valid, its gain will be  $(1 - p)(R - c_{\text{send}}) - c_{\text{check}} - p\kappa$ ; if the player deviates and does not check proposal's validity nor votes, its expected gain at deviation is  $-p\kappa$ , the gain at deviation is lower than the gain at equilibrium. The strategy profile is indeed a Nash equilibrium.
- It remains to prove that the strategy profile where some players are supposed to check the proposal validity and check only if the block is valid and the remaining players vote without checking block validity is also a Nash equilibrium.

We can first note that only valid blocks can be produced following the equilibrium, and invalid blocks do not have the necessary  $\nu$  votes, since only  $\nu - 1$  players vote without checking, and so for invalid proposal.

## 12:12 Rational Behaviors in Committee-Based Blockchains

- The expected gain of a player not supposed to check is  $(1-p)(R - c_{\text{send}})$ . If it deviates and does not vote, its gain at deviation is 0; if it deviates by checking and voting only if the proposal is valid, its expected gain at deviation is  $(1-p)(R - c_{\text{send}}) - c_{\text{check}}$ , which is lower than the gain at equilibrium.
- The expected gain of a player supposed to check is  $(1-p)(R - c_{\text{send}}) - c_{\text{check}}$ . If it deviates and does not vote, its gain at deviation is 0. If it deviates by voting without checking the proposal's validity, any block proposed will be produced, no matter its validity since  $\nu$  votes are sent in any case, so the expected gain of the deviating player is  $R - c_{\text{send}} - p\kappa$ , which is lower than the gain at equilibrium if and only if  $\kappa \geq R - c_{\text{send}} + c_{\text{check}}/p$ .

The strategy profile is indeed a Nash equilibrium.

Moreover, there is no more equilibrium. We sketch the proof by exhibiting the main other equilibrium candidates.

- Let  $x \geq 0$ . Assume by contradiction that there exists an equilibrium where  $n - \nu - x$  players check the block validity and vote only if the proposal is valid, and the remaining  $\nu + x$  players vote without checking the block validity.

That means any block proposed will be produced, since  $\nu + x \geq \nu$  players vote without checking validity. Let  $i$  be a player supposed to check. Its expected gain is  $R - c_{\text{send}} - c_{\text{check}} - p\kappa$ , while if  $i$  deviates and votes without checking proposal validity, its expected gain will be  $R - c_{\text{send}} - p\kappa$ . Contradiction, the strategy profile is not an equilibrium.

- Let  $x > 1$ . Assume by contradiction that there exists an equilibrium where  $n - \nu + x$  players check the block validity and vote only if the proposal is valid, and the remaining  $\nu - x$  players vote without checking the block validity.

Let  $i$  be a player supposed to check. Its expected gain is  $(1-p)(R - c_{\text{send}}) - c_{\text{check}}$ . If  $i$  deviates and votes without checking proposal validity, there will be  $\nu - x + 1 < \nu$  votes for invalid an block proposed, and so it will not be produced, where there will be  $n$  votes for a valid block proposed; the expected gain at deviation for  $i$  is  $(1-p)(R - c_{\text{send}})$ . Contradiction, the strategy profile proposed is not an equilibrium. ◀

► **Remark 12.** There are three types of Nash equilibria in Proposition 11.

- The equilibrium where no player votes does not guarantee *Termination* at round 1.
- In the equilibrium where no player checks, *Termination* is always guaranteed at the end of the first round, even if the proposal is invalid, which violates the *Validity* property.
- In the last equilibrium, valid blocks are produced and invalid blocks are not. *Termination* is not guaranteed at round 1 but *Validity* is always ensured. There can be at most  $\binom{n}{n-\nu+1}$  equilibria corresponding to that setting.

## 5.2 Reward All Committee Members

In this section, we consider that all committee members are rewarded once a block is produced. Equation 2 describes the gain of each rational player.

We study the different equilibria with respect to the value of  $\nu$ , the minimum number of votes required to consider a block as produced.

First, we analyze the case where 1 vote for a proposed block is sufficient to consider it as produced, *i.e.*,  $\nu = 1$ .

► **Proposition 13.** *In one round, with only rational players in the committee, if  $\nu = 1$ , when all players are rewarded once a block is produced, if there is a probability  $p$  that the proposer proposes an invalid block, and if  $\kappa \geq R - c_{\text{send}} + c_{\text{check}}/p$ , in all Nash equilibria, exactly one player checks the validity of the proposal and votes only if it is valid, while the other players do nothing.*

As in Proposition 5, one can note that in equilibrium, the task of validating (checking) and producing a block is delegated to one player.

► **Remark 14.** Note that there exists at most  $n$  equilibria corresponding to Proposition 13. In all the equilibria corresponding to Proposition 13, if the proposal is invalid, there is no Termination at the first round, however, Validity is always ensured.

We now consider the situation where strictly more than one vote is needed to consider a block as produced, *i.e.*,  $\nu \in \{2, \dots, n\}$ .

► **Proposition 15.** *In one round, with only rational players in the committee, if  $\nu > 1$ , when all players are rewarded once a block is produced, if there is a probability  $p$  that the proposer proposes an invalid block, and if  $\kappa \geq R - c_{\text{send}} + c_{\text{check}}/p$ , in all Nash equilibria, either (i) no player votes, or (ii) 1 player checks the proposal validity and votes only if it is valid, exactly  $\nu - 1$  other players vote without checking validity, and the others do nothing.*

**Proof of Proposition 15.** We prove that the strategy profiles described in the proposition are Nash equilibria.

- First, we prove that the strategy profile where no player votes is a Nash equilibrium. The gain at equilibrium of any player is 0. If one player deviates and votes, there is only 1 vote, and the block is not produced since  $\nu > 1$ , the gain at deviation is  $-c_{\text{send}} < 0$ . If the player deviates by checking block validity, it will pay the cost of checking for nothing and will have the gain  $-c_{\text{check}} - (1 - p)c_{\text{send}} < 0$ . The strategy profile is a Nash equilibrium.
- It remains to prove that the strategy profile where some players are supposed to check the proposal validity, and vote only if the block is valid; some players vote without checking block validity; and the others do nothing is a Nash equilibrium.

We first note that only valid blocks can be produced following the equilibrium, and invalid blocks do not have the necessary  $\nu$  votes, since only  $\nu - 1$  players vote without checking.

- First, the players that do not vote nor check validity have an expected gain of  $(1 - p)R$ . Let  $i$  be such a player. If  $i$  deviates and votes without checking, any proposal will be produced, no matter its validity, therefore, the gain of the player at deviation is  $R - c_{\text{send}} - p\kappa$ , which is lower than the gain at equilibrium. If instead,  $i$  deviates and checks the validity of the proposal and votes only if it is valid, only valid blocks will be produced, so the gain at deviation will be  $(1 - p)(R - c_{\text{send}}) - c_{\text{check}}$ , which is lower than the gain at equilibrium.
- Now, turns to the players not supposed to check but vote. Their expected gain at equilibrium is  $(1 - p)R - c_{\text{send}}$ . Let  $i$  be such a player, if it deviates and does not vote nor checks, no block will be produced and its gain at deviation is  $0 < (1 - p)R - c_{\text{send}}$ . If it deviates by checking and voting only if the proposal is valid, its expected gain at deviation is  $(1 - p)(R - c_{\text{send}}) - c_{\text{check}}$ , which is lower than the gain at equilibrium since  $c_{\text{send}} < c_{\text{check}}$ .
- Finally, we can analyze the one player supposed to check. Without loss of generality, assume that it is player with index 1. The expected gain of player 1 is  $(1 - p)(R - c_{\text{send}}) - c_{\text{check}}$ . If it deviates and does not vote, no block will be produced, so its gain at deviation is  $0 < (1 - p)(R - c_{\text{send}}) - c_{\text{check}}$ ; if it deviates by voting without checking the proposal's validity, any block proposed will be produced, no matter its validity since  $\nu$  votes are sent in any case; therefore, the expected gain of player 1 at deviation is  $R - c_{\text{send}} - p\kappa$ , which is lower than the gain at equilibrium if and only if  $\kappa \geq R - c_{\text{send}} + c_{\text{check}}/p$ .

The strategy profile is indeed a Nash equilibrium. No deviation is profitable.

## 12:14 Rational Behaviors in Committee-Based Blockchains

There is no more equilibrium in this setting.

First, let us note that in any case, exactly  $\nu$  players should vote (counting also those supposed to vote after checking). If there are less than  $\nu$  players supposed to vote (but at least one), no block is produced so one such player can deviate and not vote, economizing its cost. If there are more than  $\nu$  players supposed to vote, one can deviate by not voting and economizing that cost.

We can show that the other main equilibrium candidates are not equilibria.

- By contradiction, assume that there exists an equilibrium where  $\nu$  players vote without checking the proposal's validity and the others do not vote nor check.

Let  $i$  be a player supposed to vote. Its expected gain at equilibrium  $R - c_{\text{send}} - p\kappa$ , while if  $i$  deviates by checking the proposal validity and voting only if valid, only valid proposal will be produced, so its expected gain will be:  $(1 - p)(R - c_{\text{send}}) - c_{\text{check}}$  which is greater than the equilibrium. Contradiction, the strategy profile is not an equilibrium.

- By contradiction, assume that there exists an equilibrium where  $\nu$  players vote (counting also those supposed to vote after checking) and the others do not vote nor check. Suppose that in the set of players supposed to vote, at least two  $i$  and  $j$  check the validity of the proposal and vote only if it is valid. In this strategy profile, only valid proposals will be produced. The expected gain at equilibrium of  $i$  is  $(1 - p)(R - c_{\text{send}}) - c_{\text{check}}$ . If instead,  $i$  deviates and always votes without checking validity, its expected gain at deviation is  $(1 - p)R - c_{\text{send}}$ , which is greater than the gain at equilibrium. Contradiction, the strategy profile is not an equilibrium. ◀

► **Remark 16.** There are two types of Nash equilibria in Proposition 15.

- Termination is not guaranteed at round 1 in the equilibrium where no player votes.
- In the second type of equilibrium in this setting, there are exactly  $\nu$  votes when the proposal is valid, but  $\nu - 1$  votes when the proposal is invalid. Termination is not guaranteed at round 1 but Validity is ensured. There can be at most  $n * \binom{n-1}{\nu-1}$  equilibria corresponding to that setting.

A summary of the different equilibria with trembling players can be found in Table 2. When all players are rewarded once a block is produced, there is no “bad” equilibrium, *i.e.*, an equilibrium where Validity is violated, while when only players that vote for a produced block are rewarded when  $\nu < n$ , there exists a “bad” equilibrium (Propositions 9 and 11).

## 6 Discussions

Before concluding, we discuss some interesting points that are not directly addressed in the core of this paper.

**Fixed amount of Reward for the Committee.** First, we quickly highlight what happens if there is a fixed reward for the committee members that is shared by them. Let  $\mu$  be the number of players that are rewarded in the committee, and let  $\frac{R}{\mu}$  be the fraction of the reward each player rewarded gets. Our equilibrium analysis still holds, but attention should be given to the bounds. For example, in Proposition 9, instead of  $\kappa \geq R - c_{\text{send}} + c_{\text{check}}/p$  we should have  $\kappa \geq R/n - c_{\text{send}} + c_{\text{check}}/p$ , since all players vote in case of a valid proposal. (here,  $\mu = n$ ).

**Honest Players.** Recall that honest players always follow the prescribed protocol, *i.e.*, they always check the validity of the proposal, and vote only if the proposal is valid.

We did not include honest players in this paper for the following reason: their presence does not change the different equilibria we have; they may however change the bounds under which some equilibria exist.

Denote by  $h$  the number of honest players in the committee. Generally, if there are  $h$  honest players and  $\nu$  messages are required for the production of a block,  $h$  votes are guaranteed for valid blocks only; then for the rational players, the goal is to give the  $\nu - h$  remaining votes.

## 7 Conclusion

We analyze the behavior of rational players in committee-based blockchains under assumptions of synchrony of messages and under different mechanisms of rewards. Although our analysis focuses on a single round, the behaviors of the rational players can be repeated in multi-round settings, since each new round has the same setting as the first round, and therefore, be viewed as independent. This paper study the case where there is one proposer at the time and does not consider the case of multiple proposers as in Algorand which has multiple proposers at the same time and rely on probabilistic consensus while we consider only deterministic consensus. We found that although there always exist equilibria where the consensus properties are guaranteed, there also exist equilibria, where those properties are violated, both in the case where the proposal is always valid, or by trembling effect can be invalid. When all committee members are rewarded once a block is produced, in equilibrium, the validity property is always guaranteed; while rewarding only those who vote for the produced block leads to the existence of equilibrium where an invalid block can be produced. Moreover, equilibria when all committee members are rewarded are more efficient in terms of the number of messages. Thus, rewarding all members of the committee seems to be an interesting reward scheme, and need more investigations, in particular in more general settings.

In this work, we consider only the game representing the consensus agreement in committee-based blockchains, from a liveness point of view. Interestingly, it helps clearly identifying coordination problems that are likely to be present in more general settings.

---

## References

- 1 Ittai Abraham, Danny Dolev, and Joseph Y. Halpern. Distributed protocols for leader election: A game-theoretic perspective. *ACM Trans. Economics and Comput.*, 7(1), 2019.
- 2 Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. Solidus: An incentive-compatible cryptocurrency based on permissionless byzantine consensus. *CoRR*, abs/1612.02916v1, 2016.
- 3 Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. Solida: A blockchain protocol based on reconfigurable byzantine consensus. In *OPODIS 2017, Lisbon, Portugal*, 2017.
- 4 Yackolley Amoussou-Guenou, Bruno Biais, Maria Potop-Butucaru, and Sara Tucci-Piergiovanni. Rational vs byzantine players in consensus-based blockchains. In *AAMAS '20, Auckland, New Zealand*, 2020.
- 5 Yackolley Amoussou-Guenou, Antonella Del Pozzo, Maria Potop-Butucaru, and Sara Tucci-Piergiovanni. Correctness of tendermint-core blockchains. In *OPODIS 2018, Hong Kong, China*, 2018.
- 6 Bruno Biais, Christophe Bisière, Matthieu Bouvard, and Catherine Casamatta. The blockchain folk theorem. *The Review of Financial Studies*, 32(5), April 2019.

## 12:16 Rational Behaviors in Committee-Based Blockchains

- 7 Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. Technical report, Tendermint, July 2018. [arXiv:1807.04938](https://arxiv.org/abs/1807.04938).
- 8 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *OSDI'99, New Orleans, Louisiana, USA, February 22-25, 1999*, 1999.
- 9 Benjamin Y. Chan and Elaine Shi. Streamlet: Textbook streamlined blockchains. *IACR Cryptol. ePrint Arch.*, 2020.
- 10 Simon Collet, Pierre Fraigniaud, and Paolo Penna. Equilibria of games in networks for local tasks. In *OPODIS 2018, Hong Kong, China, 2018*.
- 11 Fiery Cushman, Anna Dreber, Ying Wang, and Jay Costa. Accidental Outcomes Guide Punishment in a “Trembling Hand” Game. *PLoS one*, 4(8), 2009.
- 12 Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2), 1988.
- 13 Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *CRYPTO '92, Santa Barbara, California, USA, 1992*.
- 14 Ittay Eyal and Emin Gün Sirer. Majority is not enough: bitcoin mining is vulnerable. *Commun. ACM*, 61(7), 2018.
- 15 Mehdi Fooladgar, Mohammad Hossein Manshaei, Murtuza Jadliwala, and Mohammad Ashiqur Rahman. On incentive compatible role-based reward distribution in algorand. *CoRR*, 2019.
- 16 Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *SOSP 07, Shanghai, China, 2017*.
- 17 Joseph Y. Halpern and Xavier Vilaça. Rational consensus: Extended abstract. In *PODC 2016, Chicago, IL, USA, July 25-28, 2016*.
- 18 Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *CRYPTO '17, Santa Barbara, CA, USA, 2017*.
- 19 Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3), July 1982.
- 20 Anna Lysyanskaya and Nikos Triandopoulos. Rationality and adversarial behavior in multi-party computation. In *CRYPTO 2006, Santa Barbara, California, USA, 2006*.
- 21 Jean-Philippe Martin and Lorenzo Alvisi. Fast byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, July 2006.
- 22 Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008.
- 23 John Nash. Non-cooperative games. *Annals of Mathematics*, 54(2), 1951.
- 24 Sam Toueg. Randomized byzantine agreements. In *Third Annual ACM Symposium on Principles of Distributed Computing, Vancouver, Canada, 1984*.
- 25 Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151, 2014.
- 26 Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan-Gueta, and Ittai Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *PODC 2019, Toronto, Canada, 2019*.



# Relaxed Queues and Stacks from Read/Write Operations

**Armando Castañeda**

Instituto de Matemáticas, UNAM, Mexico City, Mexico  
armando.castaneda@im.unam.mx

**Sergio Rajsbaum**

Instituto de Matemáticas, UNAM, Mexico City, Mexico  
rajsbaum@matem.unam.mx

**Michel Raynal**

Institut Universitaire de France, IRISA-Université de Rennes, France  
Polytechnic University of Hong Kong, Hong Kong  
michel.raynal@irisa.fr

---

## Abstract

Considering asynchronous shared memory systems in which any number of processes may crash, this work identifies and formally defines relaxations of queues and stacks that can be non-blocking or wait-free while being implemented using only read/write operations. Set-linearizability and Interval-linearizability are used to specify the relaxations formally, and precisely identify the subset of executions which preserve the original sequential behavior. The relaxations allow for an item to be returned more than once by different operations, but only in case of concurrency; we call such a property *multiplicity*. The stack implementation is wait-free, while the queue implementation is non-blocking. Interval-linearizability is used to describe a queue with multiplicity, with the additional relaxation that a dequeue operation can return *weak-empty*, which means that the queue *might* be empty. We present a read/write wait-free interval-linearizable algorithm of a concurrent queue. As far as we know, this work is the first that provides formalizations of the notions of multiplicity and weak-emptiness, which can be implemented on top of read/write registers only.

**2012 ACM Subject Classification** Theory of computation → Distributed computing models; Computing methodologies → Distributed algorithms; Computing methodologies → Concurrent algorithms; Theory of computation → Concurrent algorithms; Theory of computation → Distributed algorithms

**Keywords and phrases** Asynchrony, Correctness condition, Linearizability, Nonblocking, Process crash, Relaxed data type, Set-linearizability, Wait-freedom, Work-stealing

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2020.13

**Related Version** A full version of the paper is available at [15]. <https://arxiv.org/abs/2005.05427>.

**Funding** *Armando Castañeda*: Supported by UNAM-PAPIIT project IN108720.

*Sergio Rajsbaum*: Supported by UNAM-PAPIIT project IN106520.

*Michel Raynal*: Supported by French ANR project DESCARTES (16-CE40-0023-03).

## 1 Introduction

In the context of asynchronous crash-prone systems where processes communicate by accessing a shared memory, linearizable implementations of concurrent counters, queues, stacks, pools, and other concurrent data structures [32] need extensive synchronization among processes, which in turn jeopardizes performance and scalability. Moreover, it has been formally shown that this cost is sometimes unavoidable, under various specific assumptions [11, 12, 19]. However, often applications do not require all guarantees offered by a linearizable sequential specification [38]. Thus, much research has focused on improving performance of concurrent



© Armando Castañeda, Sergio Rajsbaum, and Michel Raynal;  
licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 13; pp. 13:1–13:19

Leibniz International Proceedings in Informatics



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

data structures by relaxing their semantics. Furthermore, several works have focused on relaxations for queues and stacks, achieving significant performance improvements (e.g., [21, 22, 28, 38]).

It is impossible however to implement queues and stacks with only Read/Write operations, without relaxing their specification. This is because queues and stacks have consensus number two (i.e. they allow consensus to be solved among two processes but not three), while the consensus number of Read/Write operations is only one [23], hence too weak to wait-free implement queues and stacks. Thus, atomic Read-Modify-Write operations, such as Compare&Swap or Test&Set, are required in any queue or stack implementation. To the best of our knowledge, even relaxed versions of queues or stacks have not been designed that avoid the use of Read-Modify-Write operations.

In this article, we are interested in exploring if there are meaningful relaxations of queues and stacks that can be implemented using only simple Read/Write operations, namely, if there are non-trivial relaxations with consensus number one. Hence, this work is a theoretical investigation of the power of the crash Read/Write model for relaxed data structures.

## Contributions

We identify and formally define relaxations of queues and stacks that can be implemented using only simple Read/Write operations. We consider queue and stack relaxations with *multiplicity*, where an item can be extracted by more than one dequeue or pop operation, instead of exactly once. However, this may happen only in the presence of concurrent operations. As already argued [30], this type of relaxation could be useful in a wide range of applications, such as parallel garbage collection, fixed point computations in program analysis, constraint solvers (e.g. SAT solvers), state space search exploration in model checking, as well as integer and mixed programming solvers.

One of the main challenges in designing relaxed data structures lies in the difficulty of formally specifying what is meant by “relaxed specification”. To provide a formal specification of our relaxations, we use *set-linearizability* [33] and *interval-linearizability* [14], specification methods that are useful to specify the behavior of a data structure in concurrent patterns of operation invocations, instead of only in sequential patterns. Using these specification methods, we are able to precisely state in which executions the relaxed behavior of the data structure should take place, and demand a strict behavior (not relaxed), in other executions, especially when operation invocations are sequential.

**First Contribution.** We define a *set-concurrent stack with multiplicity*, in which no items are lost, all items are pushed/popped in LIFO order but an item can be popped by multiple operations, which are then concurrent. We define a *set-concurrent queue with multiplicity* similarly. In both cases we present set-linearizable implementations based only on Read/Write operations. The stack implementation is wait-free [23], while the queue implementation is non-blocking [25]. Our set-concurrent implementations imply Read/Write solutions for *idempotent work-stealing* [30] and *k-FIFO* [28] queues and stacks.

**Second Contribution.** We define an interval-concurrent queue with a *weak-emptiness check*, which behaves like a classical sequential queue with the exception that a dequeue operation can return a control value denoted *weak-empty*. Intuitively, this value means that the operation was concurrent with dequeue operations that took the items that were in the queue when it started, thus the queue might be empty. First, we describe a wait-free interval-linearizable implementation based on Fetch&Inc and Swap operations. Then, using the techniques in our set-linearizable stack and queue implementations, we obtain a wait-free interval-linearizable implementation using only Read/Write operations.



Our interval-concurrent queue with weak-emptiness check is motivated by a theoretical question that has been open for more than two decades [4]: it is unknown if there is a wait-free linearizable queue implementation based on objects with consensus number two (e.g. `Fetch&Inc` or `Swap`), for any number of processes. There are only such non-blocking implementations in the literature, or wait-free implementations for restricted cases (e.g. [10, 18, 29, 16, 17]). Interestingly, our interval-concurrent queue allows us to go from non-blocking to wait-freedom.

Since we are interested in the computability power of Read/Write operations to implement relaxed concurrent objects (that otherwise are impossible), our algorithms are presented in an idealized shared-memory computational model. We hope these algorithms will help to develop a better understanding of fundamentals that can derive solutions for real multicore architectures, with good performance and scalability.

## Related Work

It has been frequently pointed out that classic concurrent data structures have to be relaxed in order to support scalability, and examples are known showing how natural relaxations on the ordering guarantees of queues or stacks can result in higher performance and greater scalability [38]. Thus, for the past ten years there has been a surge of interest in relaxed concurrent data structures from practitioners (e.g. [34]). Also, theoreticians have identified inherent limitations in achieving high scalability in the implementation of linearizable objects [11, 12, 19].

Some articles relax the sequential specification of traditional data structures, while others relax their correctness condition requirements. As an example of relaxing the requirement of a sequential data structure, [22, 27, 28, 35] present a *k-FIFO* queue (called *out-of-order* in [22]) in which elements may be dequeued out of FIFO order up to a constant  $k \geq 0$ . A family of relaxed queues and stacks is introduced in [39], and studied from a computability point of view (consensus numbers). It is defined in [22] the *k-stuttering* relaxation of a queue/stack, where an item can be returned by a dequeue/pop operation without actually removing the item, up to  $k \geq 0$  times, even in sequential executions. Our queue/stack with multiplicity is a stronger version of *k-stuttering*, in the sense that an item can be returned by two operations if and only if the operations are concurrent. Relaxed priority queues (in the flavor of [39]) and associated performance experiments are presented in [6, 42].

Other works design a weakening of the consistency condition. For instance, *quasi-linearizability* [3], which models relaxed data structures through a distance function from valid sequential executions. This work provides examples of quasi-linearizable concurrent implementations that outperform state of the art standard implementations. A *quantitative* relaxation framework to formally specify relaxed objects is introduced in [21, 22] where relaxed queues, stacks and priority queues are studied. This framework is more powerful than quasi-linearizability. It is shown in [40] that linearizability and three data type relaxations studied in [22], *k-Out-of-Order*, *k-Lateness*, and *k-Stuttering*, can also be defined as consistency conditions. The notion of *local linearizability* is introduced in [20]. It is a relaxed consistency condition that is applicable to container-type concurrent data structures like pools, queues, and stacks. The notion of *distributional linearizability* [5] captures *randomized* relaxations. This formalism is applied to MultiQueues [37], a family of concurrent data structures implementing relaxed concurrent priority queues.

The previous works use relaxed specifications, but still sequential, while we relax the specification to make it concurrent (using set-linearizability and interval-linearizability).

The notion of *idempotent work stealing* is introduced in [30], where LIFO, FIFO and double-ended set implementations are presented; these implementations exploit the relaxed semantics to deliver better performance than usual work stealing algorithms. Similarly

to our queues and stacks with multiplicity, the *idempotent* relaxation means that each inserted item is eventually extracted at least once, instead of exactly once. In contrast to our work, the algorithms presented in [30] use **Compare&Swap** (in the **Steal** operation). Being a practical-oriented work, formal specifications of the implemented data structures are not given.

**Organization.** The article is organized as follows. Section 2 presents the model of computation and the linearizability, set-linearizability and interval-linearizability correctness conditions. Sections 3 and 4 contain our **Read/Write** set-linearizable queue and stack implementations, and Section 5 explains some implications obtained from these implementations. Our interval-linearizable queue implementation is presented in Section 6. Finally, Section 7 ends the paper with some final remarks. Full proofs of our claims can be found in the full version of the paper [15].

## 2 Preliminaries

**Model of Computation.** We consider the standard concurrent system model with  $n$  *asynchronous* processes,  $p_1, \dots, p_n$ , which may *crash* at any time during an execution. The *index* of process  $p_i$  is  $i$ . Processes communicate with each other by invoking *atomic* operations on shared *base objects*. A base object can provide atomic **Read/Write** operations (henceforth called a *register*), or more powerful atomic **Read-Modify-Write** operations, such as **Fetch&Inc**, **Swap** or **Compare&Swap**.

A (*high-level*) *concurrent object*, or *data type*, is, roughly speaking, defined by a state machine consisting of a set of states, a finite set of operations, and a set of transitions between states. The specification does not necessarily have to be *sequential*, namely, (1) a state might have pending operations and (2) state transitions might involve several invocations. The following subsections formalize this notion and the different types of objects.

An *implementation* of a concurrent object  $T$  is a distributed algorithm  $\mathcal{A}$  consisting of local state machines  $A_1, \dots, A_n$ . Local machine  $A_i$  specifies which operations on base objects  $p_i$  executes in order to return a response when it invokes a high-level operation of  $T$ . A process is *sequential*: it can invoke a new high-level operations only when its previous operation has been responded. Each of these base objects operation invocations is a *step*. Thus, an *execution* of  $\mathcal{A}$  is a possibly infinite sequence of steps, namely, executions of base objects operations, plus invocations and responses to high-level operations of the concurrent object  $T$ .

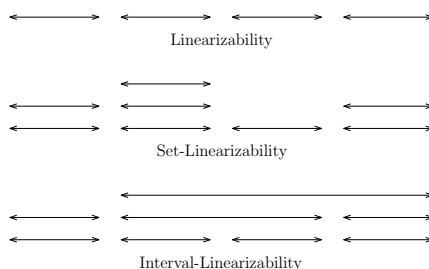
An operation in an execution is *complete* if both its invocation and response appear in the execution. An operation is *pending* if only its invocation appears in the execution. A process is *correct* in an execution if it takes infinitely many steps.

An implementation is *wait-free* if every process completes each operation it invokes [23]. An implementation is *non-blocking* if whenever processes take steps and at least one of them does not crash, at least one of them terminates its operation [25]. Thus, a wait-free implementation is non-blocking but not necessarily vice versa.

The *consensus number* of a shared object  $O$  is the maximum number of processes that can solve *consensus*, using any number of instances of  $O$  in addition to any number of **Read/Write** registers [23]. Consensus numbers induce the *consensus hierarchy* where objects are classified according their consensus numbers. The simple **Read/Write** operations stand at the bottom of the hierarchy, with consensus number one; these operations are the least expensive ones in real multicore architectures. At the top of the hierarchy we find operations with infinite consensus number, like **Compare&Swap**, that provide the maximum possible coordination.

**Correctness Conditions.** *Linearizability* [25] is the standard notion used to define a correct concurrent implementation of an object defined by a sequential specification. Intuitively, an execution is linearizable if its operations can be totally ordered, while respecting the execution order its non-concurrent operations (see below).

A *sequential specification* of a concurrent object  $T$  is a state machine specified through a transition function  $\delta$ . Given a state  $q$  and an invocation  $inv(\text{op})$ ,  $\delta(q, inv(\text{op}))$  returns the tuple  $(q', res(\text{op}))$  (or a set of tuples if the machine is *non-deterministic*) indicating that the machine moves to state  $q'$  and the response to  $\text{op}$  is  $res(\text{op})$ . In our specifications,  $res(\text{op})$  is written as a tuple  $\langle \text{op} : r \rangle$ , where  $r$  is the output value of the operation. The sequences of invocation-response tuples,  $\langle inv(\text{op}) : res(\text{op}) \rangle$ , produced by the state machine are its *sequential executions*.



■ **Figure 1** Linearizability requires a total order on the operations, set-linearizability allows several operations to be linearized at the same linearization point, while interval-linearizability allows an operation to be decomposed into several linearization points.

To formalize linearizability we define a partial order  $<_{\alpha}$  on the completed operations of an execution  $\alpha$ :  $\text{op} <_{\alpha} \text{op}'$  if and only if  $res(\text{op})$  precedes  $inv(\text{op}')$  in  $\alpha$ . Two operations are *concurrent*, denoted  $\text{op} \parallel_{\alpha} \text{op}'$ , if they are incomparable by  $<_{\alpha}$ . The execution is *sequential* if  $<_{\alpha}$  is a total order.

Let  $\mathcal{A}$  be an implementation of a concurrent object  $T$ . An execution  $\alpha$  of  $\mathcal{A}$  is *linearizable* if there is a sequential execution  $S$  of  $T$  such that: (1)  $S$  contains every completed operation of  $\alpha$  and might contain some pending operations. Inputs and outputs of invocations and responses in  $S$  agree with inputs and outputs in  $\alpha$ . (2) For every two completed operations  $\text{op}$  and  $\text{op}'$  in  $\alpha$ , if  $\text{op} <_{\alpha} \text{op}'$ , then  $\text{op}$  appears before  $\text{op}'$  in  $S$ . We say that  $\mathcal{A}$  is *linearizable* if each of its executions is linearizable.

To formally specify our relaxed queues and stacks, we use the formalism provided by the set-linearizability and interval-linearizability consistency conditions [14, 33]. Roughly speaking, set-linearizability allows us to linearize several operations in the same point, namely, all these operations are executed concurrently, while interval-linearizability allows operations to be linearized concurrently with several non-concurrent operations. Figure 1 schematizes the differences between the three consistency conditions where each double-end arrow represents an operation execution.

A *set-concurrent specification* of a concurrent object differs from a sequential execution in that  $\delta$  receives as input the current state  $q$  of the machine and a set  $Inv = \{inv(\text{op}_1), \dots, inv(\text{op}_t)\}$  of operation invocations, and  $\delta(q, Inv)$  returns  $(q', Res)$ , where  $q'$  is the next state and  $Res = \{res(\text{op}_1), \dots, res(\text{op}_t)\}$  are the responses to the invocations in  $Inv$ . Intuitively, all operations  $\text{op}_1, \dots, \text{op}_t$  are performed concurrently and move the machine from state  $q$  to  $q'$ . The sets  $Inv$  and  $Res$  are called *concurrency classes*. Observe that a set-concurrent specification in which all concurrency classes have a single element corresponds to a sequential specification.

Let  $\mathcal{A}$  be an implementation of a concurrent object  $T$ . An execution  $\alpha$  of  $\mathcal{A}$  is *set-linearizable* if there is a set-concurrent execution  $S$  of  $T$  such that: (1)  $S$  contains every completed operation of  $\alpha$  and might contain some pending operations. Inputs and outputs of invocations and responses in  $S$  agree with inputs and outputs in  $\alpha$ . (2) For every two completed operations  $\text{op}$  and  $\text{op}'$  in  $\alpha$ , if  $\text{op} <_{\alpha} \text{op}'$ , then  $\text{op}$  appears before  $\text{op}'$  in  $S$ . We say that  $\mathcal{A}$  is *set-linearizable* if each of its executions is set-linearizable.

In an *interval-concurrent specification*, some operations might be pending in a given state  $q$ , namely, the state records that there is an operation of a process without response. We now have that in  $(q', \text{Res}) = \delta(q, \text{Inv})$ , some of the operations that are pending in  $q$  might still be pending in  $q'$  and operations invoked in  $\text{Inv}$  may be pending in  $q'$ , therefore  $\text{Res}$  contains the responses to the operations that are completed when moving from  $q$  to  $q'$ .

Let  $\mathcal{A}$  be an implementation of a concurrent object  $T$ . An execution  $\alpha$  of  $\mathcal{A}$  is *interval-linearizable* if there is an interval-concurrent execution  $S$  of  $T$  such that: (1)  $S$  contains every completed operation of  $\alpha$  and might contain some pending operations. Inputs and outputs of invocations and responses in  $S$  agree with inputs and outputs in  $\alpha$ . (2) For every two completed operations  $\text{op}$  and  $\text{op}'$  in  $\alpha$ , if  $\text{op} <_{\alpha} \text{op}'$ , then  $\text{op}$  appears before  $\text{op}'$  in  $S$ . We say that  $\mathcal{A}$  is *interval-linearizable* if each of its executions is interval-linearizable.

### 3 Set-Concurrent Stacks with Multiplicity

By the *universality* of consensus [23], we know that, for every sequential object there is a linearizable wait-free implementation of it, for any number of processes, using Read/Write registers and base objects with consensus number  $\infty$ , e.g. Compare&Swap [24, 36, 41]. However, the resulting implementation might not be efficient because first, as it is universal, the construction does not exploit the semantics of the particular object, and Compare&Swap may be an expensive base operation. Moreover, such an approach would prevent us from investigating the power and the limit of the Read/Write model (as it was done for Snapshot object for which there are several linearizable wait-free Read/Write efficient implementations, e.g. [1, 8, 26]) and find accordingly meaningful Read/Write-based specifications of relaxed sequential specifications with efficient implementations.

**A Wait-free Linearizable Stack from Consensus Number Two.** Afek, Gafni and Morisson proposed in [2] a simple linearizable wait-free stack implementation for  $n \geq 2$  processes, using Fetch&Inc and Test&Set base objects, whose consensus number is 2. Figure 2 contains a slight variant of this algorithm that uses Swap and *readable* Fetch&Inc objects, both with consensus number 2 (the authors explain in [2] how to replace Test&Set with Swap).

A Push operation reserves a slot in *Item* by atomically reading and incrementing *Top* (Line 01) and then places its item in the corresponding position (Line 02). A Pop operation simply reads the *Top* of the stack (Line 04) and scans down *Items* from that position (Line 05), trying to obtain an item with the help of a Swap operation (Lines 06 and 07); if the operation cannot get a item (a non- $\perp$  value), it returns empty (Line 09). In what follows, we call this implementation Seq-Stack. It is worth mentioning that, although Seq-Stack has a simple structure, its linearizability proof is far from trivial, the difficult part being proving that items are taken in LIFO order.

In a formal sense, Seq-Stack is the best we can do, from the perspective of the consensus hierarchy: if there were a wait-free (or non-blocking) linearizable implementation based only on Read/Write registers, we could solve consensus among two processes in the standard way, by popping a value from the stack initialized to a single item containing a predefined value `winner`;

<p><b>Shared Variables:</b>  <math>Top</math> : Fetch&amp;Inc object initialized to 1  <math>Items[1, \dots]</math> : Swap objects initialized to <math>\perp</math></p> <p><b>Operation Push(<math>x_i</math>) is</b>  (01) <math>top_i \leftarrow Top.Fetch\&amp;Inc()</math>  (02) <math>Items[top_i].Write(x_i)</math>  (03) return true  end Push</p> <p><b>Operation Pop() is</b>  (04) <math>top_i \leftarrow Top.Read() - 1</math>  (05) for <math>r_i \leftarrow top_i</math> down to 1 do  (06) <math>x_i \leftarrow Items[r_i].Swap(\perp)</math>  (07) if <math>x_i \neq \perp</math> then return <math>x_i</math> end if  (08) end for  (09) return <math>\epsilon</math>  end Pop</p>
---

■ **Figure 2** Stack implementation Seq-Stack of Afek, Gafni and Morisson [2] (code for process  $p_i$ ).

this is a contradiction as consensus cannot be solved from Read/Write registers [24, 36, 41]. Therefore, there is no *exact* wait-free linearizable stack implementation from Read/Write registers only. However, we could search for *approximate* solutions. Below, we show a formal definition of the notion of a *relaxed* set-concurrent stack and prove that it can be wait-free implemented from Read/Write registers. Informally, our solution consists in implementing relaxed versions of Fetch&Inc and Swap with Read/Write registers, and plug these implementations in Seq-Stack.

**A Set-linearizable Read/Write Stack with Multiplicity.** Roughly speaking, our relaxed stack allows concurrent Pop operations to obtain the same item, but all items are returned in LIFO order, and no pushed item is lost. Formally, our set-concurrent stack is specified as follows:

► **Definition 1** (Set-Concurrent Stack with Multiplicity). *The universe of items that can be pushed is  $\mathbf{N} = \{1, 2, \dots\}$ , and the set of states  $Q$  is the infinite set of strings  $\mathbf{N}^*$ . The initial state is the empty string, denoted  $\epsilon$ . In state  $q$ , the first element in  $q$  represents the top of the stack, which might be empty if  $q$  is the empty string. The transitions are the following:*

1. For  $q \in Q$ ,  $\delta(q, \text{Push}(x)) = (x \cdot q, \langle \text{Push}(x) : \text{true} \rangle)$ .
2. For  $q \in Q$ ,  $1 \leq t \leq n$  and  $x \in \mathbf{N}$  :  $\delta(x \cdot q, \{\text{Pop}_1(), \dots, \text{Pop}_t()\}) = (q, \{\langle \text{Pop}_1() : x \rangle, \dots, \langle \text{Pop}_t() : x \rangle\})$ .
3.  $\delta(\epsilon, \text{Pop}()) = (\epsilon, \langle \text{Pop}() : \epsilon \rangle)$ .

► **Remark 2.** Every execution of the set-concurrent stack with all its concurrency classes containing a single operation boils down to an execution of a sequential stack.

The following lemma shows that any algorithm implementing the set-concurrent stack keeps the behavior of a sequential stack in several cases. In fact, the only reason the implementation does not provide linearizability is due only to the Pop operations that are concurrent.

► **Lemma 3.** *Let  $A$  be any set-linearizable implementation of the set-concurrent stack with multiplicity. Then,*

1. *All sequential executions of  $A$  are executions of the sequential stack.*
2. *All executions with no concurrent Pop operations are linearizable with respect to the sequential stack.*

3. All executions with Pop operations returning distinct values are linearizable with respect to the sequential stack.
4. If Pop operations return the same value in an execution, then they are concurrent.

The algorithm in Figure 3 is a set-linearizable Read/Write wait-free implementation of the stack with multiplicity, which we call **Set-Conc-Stack**. This implementation is a modification of **Seq-Stack**. The **Fetch&Inc** operation in Line 01 in **Seq-Stack** is replaced by a **Read** and **Increment** operations of a Read/Write wait-free linearizable Counter, in Lines 01 and 02 in **Set-Conc-Stack**. This causes a problem as two **Push** operations can set the same value in their  $top_i$  local variables. This problem is resolved with the help of a two-dimensional array *Items* in Line 03, which guarantees that no pushed item is lost: each row of *Items* now has  $n$  entries, each of them associated with one and only process. Similarly, the **Swap** operation in Line 06 in **Seq-Stack** is replaced by **Read** and **Write** operations in Lines 08 and 10 in **Set-Conc-Stack**, together with the test in Line 09 which ensures that a **Pop** operation modifies an entry in *Items* only if an item has been written in it. Thus, it is now possible that two distinct **Pop** operations get the same non- $\perp$  value, which is fine because this can only happen if the operations are concurrent. Object *Top* in **Set-Conc-Stack** can be any of the known Read/Write wait-free linearizable Counter implementations<sup>1</sup>.

```

Shared Variables:
  Top : Read/Write Counter object initialized to 1
  Items[1, ..., n][1, ..., n] : Read/Write registers init. to  $\perp$ 

Operation Push(x) is
(01)  $top_i \leftarrow Top.Read()$ 
(02)  $Top.Increment()$ 
(03)  $Items[top_i, i].Write(x)$ 
(04) return true
end Push

Operation Pop() is
(05)  $top_i \leftarrow Top.Read() - 1$ 
(06) for  $r_i \leftarrow top_i$  down to 1 do
(07)   for  $s_i \leftarrow n$  down to 1 do
(08)      $x_i \leftarrow Items[r_i][s_i].Read()$ 
(09)     if  $x_i \neq \perp$  then
(10)        $Items[r_i][s_i].Write(\perp)$ 
(11)       return  $x_i$ 
(12)     end if
(13)   end for
(14) end for
(15) return  $\epsilon$ 
end Pop

```

■ **Figure 3** Read/Write wait-free set-concurrent stack **Set-Conc-Stack** with multiplicity (code for process  $p_i$ ).

► **Theorem 4.** *The algorithm **Set-Conc-Stack** (Figure 3) is a Read/Write wait-free set-linearizable implementation of the stack with multiplicity.*

**Proof sketch.** The set-linearizability proof is a “reduction” that proceeds as follows. For any execution  $E$ , we modify it and remove some of its operations to obtain another execution  $G$  of the algorithm. Then, from  $G$ , we obtain an execution  $H$  of **Seq-Stack**, and show that we can obtain a set-linearization  $\text{SetLin}(G)$  of  $G$  from any linearization  $\text{Lin}(H)$  of  $H$ . Finally, we add to  $\text{SetLin}(G)$  the operations of  $E$  that were removed to obtain a set-linearization

<sup>1</sup> To the best of our knowledge, the best implementation is in [7] with polylogarithmic step complexity, on the number of processes, provided that the number of increments is polynomial.



$\text{SetLin}(E)$  of  $E$ . To obtain the execution  $G$ , we first obtain intermediate executions  $F$  and then  $F'$ , from which we derive  $G$ . For any value  $y \neq \epsilon$  that is returned by at least two concurrent  $\text{Pop}$  operations in  $E$ , we remove all these operations (invocations, responses and steps) except for the first one that executes Line 10, i.e., the first among these operations that marks  $y$  as taken in  $\text{Items}$ . Let  $F$  be the resulting execution of  $\text{Set-Conc-Stack}$ . Since there are no two  $\text{Pop}$  operations in  $F$  popping the same item  $y \neq \epsilon$ , then for every  $\text{Pop}$  operation we can safely move backward each of its steps in Line 10 next to its previous step in Line 08 (which corresponds to the same iteration of the for loop in Line 07). Thus, for every  $\text{Pop}$  operation, Lines 08 to 10 correspond to a  $\text{Swap}$  operation. Let  $F'$  denote the resulting equivalent execution.

	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
4	$\perp$	$x_1^4$	$\dots$	$x_2^4$	$\dots$	$x_3^4$	$\perp$	8	$x_3^4$
3	$x_1^3$	$\perp$	$\dots$	$\dots$	$\dots$	$x_2^3$	$\perp$	7	$x_2^4$
2	$\perp$	$\perp$	$\dots$	$\dots$	$\dots$	$\perp$	$\perp$	6	$x_1^4$
1	$\perp$	$x_1^1$	$\perp$	$\dots$	$x_2^1$	$x_3^1$	$\perp$	5	$x_2^3$
	1	2	$\dots$	$\dots$	$\dots$	$\dots$	$n$	4	$x_1^3$
								3	$x_3^1$
								2	$x_2^1$
								1	$x_1^1$

■ **Figure 4** An example of the codification of the one-dimensional array  $\text{Items}$  of  $\text{Seq-Stack}$  in the two-dimensional array  $\text{Items}$  in  $\text{Set-Conc-Stack}$ . The untouched entries are represented with  $\perp$ .

We now permute the order of some steps in  $F'$  to obtain  $G$ . For each integer  $b \geq 0$ , let  $t(b) \in [0, \dots, n]$  be the number of  $\text{Push}$  operations in  $F'$  that store their items in row  $\text{Items}[b]$ . Namely, each of these operations obtains  $b$  in its  $\text{Read}$  steps in Line 01. Let  $\text{Push}_1^b, \dots, \text{Push}_{t(b)}^b$  denote all these operations. For each  $\text{Push}_j^b$ , let  $x_j^b$  denote the item the operation pushes, let  $e_j^b$  denote its  $\text{Read}$  step in Line 01, and let  $\text{ind}_j^b$  be the index of the process that performs operation  $\text{Push}_j^b$ . Hence,  $\text{Push}_j^b$  stores its item  $x_j^b$  in  $\text{Items}[b][\text{ind}_j^b]$  when performs Line 03. Without loss of generality, let us suppose that  $\text{ind}_1^b < \text{ind}_2^b < \dots < \text{ind}_{t(b)}^b$ . Observe that  $\text{Push}_1^b, \dots, \text{Push}_{t(b)}^b$  are concurrent.

Let  $f^b$  be the first among the steps  $e_1^b, \dots, e_{t(b)}^b$  that appears in  $F'$ . As explained in the full proof, moving forward each  $e_j^b$  right after  $f^b$  produces another execution equivalent to  $F'$ . Thus, we obtain  $G$  by moving forward all steps  $e_1^b, \dots, e_{t(b)}^b$  up to the position of  $f^b$ , and place them in that order,  $e_1^b, \dots, e_{t(b)}^b$ , for every  $b \geq 0$ . Intuitively, all  $\text{Push}_1^b, \dots, \text{Push}_{t(b)}^b$  concurrently read  $\text{Top}$  and then concurrently increment it.

The main observation now is that  $G$  already corresponds to an execution of  $\text{Seq-Stack}$ , if we consider the entries in  $\text{Items}$  in their usual order (first row, then column). We say that  $\text{Items}[r][s]$  is *touched* in  $G$  if there is a  $\text{Push}$  operation that writes its item in that entry; otherwise,  $\text{Items}[r][s]$  is *untouched*. Now, for every  $b \geq 0$ , in  $G$  all  $\text{Push}_1^b, \dots, \text{Push}_{t(b)}^b$  execute Line 01 one right after the other, in order  $e_1^b, \dots, e_{t(b)}^b$ . Also, the items they push appear in row  $\text{Items}[b]$  from left to right in order  $\text{Push}_1^b, \dots, \text{Push}_{t(b)}^b$ . Thus, we can think of the touched entries in row  $\text{Items}[b]$  as a column with the left most element at the bottom, and pile all rows of  $\text{Items}$  with  $\text{Items}[0]$  at the bottom. Figure 4 depicts an example of the transformation. In this way, each  $e_j^b$  corresponds to a  $\text{Fetch\&Inc}$  operation and every  $\text{Pop}$  operations scans the touched entries of  $\text{Items}$  in the order  $\text{Seq-Stack}$  does (note that it does not matter if the operation start scanning in a row of  $\text{Items}$  with no touched entries, since untouched entries are immaterial). Thus, from  $G$  we can obtain an execution  $H$  of  $\text{Seq-Stack}$ .

## 13:10 Relaxed Queues and Stacks from Read/Write Operations

Any linearization  $\text{Lin}(H)$  of  $H$  is indeed a set-linearization of  $F$  and  $G$  with each concurrency class having a single operation. To obtain a set-linearization  $\text{SetLin}(E)$  of  $E$ , we put every **Pop** operation of  $E$  that is removed to obtain  $F$ , in the concurrency class of  $\text{Lin}(H)$  with the **Pop** operation that returns the same item. Therefore,  $E$  is set-linearizable. ◀

It is worth observing that indeed it is simple to prove that **Set-Conc-Stack** is an implementation of the *set-concurrent pool with multiplicity*, namely, Definition 1 without LIFO order (i.e.  $q$  is a set instead of a string). The hard part in the previous proof is the LIFO order, which is shown through a reduction to the (nontrivial) linearizability proof of **Seq-Stack** [2].

**A Renaming-based Performance-related Improvement.** When the contention on the shared memory accesses is small, a **Pop** operation in **Set-Conc-Stack** might perform several “useless” **Read** operations in Line 08, as it scans all entries of *Items* in every row while trying to get a non- $\perp$  value, and some of these entries might never store an item in the execution (called untouched in the proof of Theorem 4). This issue can be mitigated with the help of an array *Ren* with instances of any **Read/Write**  $f(n)$ -adaptive renaming. In *f(n)-adaptive renaming* [9], each process starts with its index as input and obtains a unique name in the space  $\{1, \dots, f(p)\}$ , where  $p$  denotes the number of processes participating in the execution. Several adaptive renaming algorithms have been proposed (see e.g. [13]); a good candidate is the simple  $(p^2/2)$ -adaptive renaming algorithm of Moir and Anderson with  $O(p)$  individual step complexity [31].

Push operations storing their items in the same row  $\text{Items}[b]$ , which has now infinite length, dynamically decide where in the row they store their items, with the help of  $\text{Ren}[b].\text{Rename}(\cdot)$  before performing Line 02. Additionally, these operations announce the number of operations that store values in row  $\text{Items}[b]$  by incrementing a counter  $\text{NOPS}[b]$  before incrementing  $\text{Top}$  in Line 02. In this way, a **Pop** operation first reads the value  $x$  of  $\text{NOPS}[r_i]$  before the **for** loop in Line 06, and then scans only that segment of  $\text{Items}[r_i]$  in the **for** loop in Line 07, namely,  $\text{Item}[r_i][1, \dots, f(x)]$ .

Note that if the contention is small, say  $O(\log^x n)$ , every **Pop** operation scans only the first entries  $O(\log^{2x} n)$  of row  $\text{Items}[b]$  as the processes storing items in that row rename in the space  $\{1, \dots, (\log^{2x} n)/2\}$ , using the Moir and Anderson  $(p^2/2)$ -adaptive renaming algorithm. Finally, observe that  $n$  does not to be known in the modified algorithm (as in **Seq-Stack**).

### 4 Set-Concurrent Queues with Multiplicity

We now consider the linearizable queue implementation in Figure 5, which uses objects with consensus number two. The idea of the implementation, which we call **Seq-Queue**, is similar to that of **Seq-Stack** in the previous section. Differently from **Seq-Stack**, whose operations are wait-free, **Seq-Queue** has a wait-free **Enqueue** and a non-blocking **Dequeue**.

**Seq-Queue** is a slight modification of the non-blocking queue implementation of Li [29], which in turn is a variation of the blocking queue implementation of Herlihy and Wing [25]. Each **Enqueue** operation simply reserves a slot for its item by performing **Fetch&Inc** to the tail of the queue, Line 01, and then stores it in *Items*, Line 02. A **Dequeue** operation repeatedly tries to obtain an item scanning *Items* from position 1 to the tail of the queue (from its perspective), Line 07; every time it sees an item has been stored in an entry of *Items*, Lines 09 and 10, it tries to obtain the item by atomically replacing it with  $\top$ , which signals that the item stored in that entry has been taken, Line 11. While scanning, the operation records the number of items that has been taken (from its perspective), Line 13, and if this



number is equal to the number of items that were taken in the previous scan, it declares the queue is empty, Line 16. Despite its simplicity, Seq-Queue's linearizability proof is far from trivial.

```

Shared Variables:
  Tail : Fetch&Inc object initialized to 1
  Items[1,...] : Swap objects initialized to  $\perp$ 

Operation Enqueue( $x_i$ ) is
(01)  $tail_i \leftarrow Tail.Fetch\&Inc()$ 
(02)  $Items[tail_i].Write(x_i)$ 
(03) return true
end Enqueue

Operation Dequeue() is
(04)  $taken'_i \leftarrow 0$ 
(05) while true do
(06)    $taken_i \leftarrow 0$ 
(07)    $tail_i \leftarrow Tail.Read() - 1$ 
(08)   for  $r_i \leftarrow 1$  up to  $tail_i$  do
(09)      $x_i \leftarrow Items[r_i].Read()$ 
(10)     if  $x_i \neq \perp$  then
(11)        $x_i \leftarrow Items[r_i].Swap(\top)$ 
(12)       if  $x_i \neq \top$  then return  $x_i$  end if
(13)        $taken_i \leftarrow taken_i + 1$ 
(14)     end if
(15)   end for
(16)   if  $taken_i = taken'_i$  then return  $\epsilon$ 
(17)    $taken'_i \leftarrow taken_i$ 
(18) end while
end Dequeue

```

■ **Figure 5** Non-blocking linearizable queue Seq-Queue from base objects with consensus number 2 (code for  $p_i$ ).

Similarly to the case of the stack, Seq-Queue is optimal from the perspective of the consensus hierarchy as there is no non-blocking linearizable queue implementation from Read/Write operations only. However, as we will show below, we can obtain a Read/Write non-blocking implementation of a set-concurrent queue with multiplicity.

► **Definition 5** (Set-Concurrent Queue with Multiplicity). *The universe of items that can be enqueued is  $\mathbf{N} = \{1, 2, \dots\}$ , and the set of states  $Q$  is the infinite set of strings  $\mathbf{N}^*$ . The initial state is the empty string, denoted  $\epsilon$ . In state  $q$ , the first element in  $q$  represents the head of the queue, which might be empty if  $q$  is the empty string. The transitions are the following:*

1. For  $q \in Q$ ,  $\delta(q, \text{Enqueue}(x)) = (q \cdot x, \langle \text{Enqueue}(x) : \text{true} \rangle)$ .
2. For  $q \in Q$ ,  $1 \leq t \leq n$ ,  $x \in \mathbf{N}$ :  $\delta(x \cdot q, \{\text{Dequeue}_1(), \dots, \text{Dequeue}_t()\}) = (q, \{\langle \text{Dequeue}_1() : x \rangle, \dots, \langle \text{Dequeue}_t() : x \rangle\})$ .
3.  $\delta(\epsilon, \text{Dequeue}()) = (\epsilon, \langle \text{Dequeue}() : \epsilon \rangle)$ .

► **Remark 6.** Every execution of the set-concurrent queue with all its concurrency classes containing a single operation is an execution of the sequential queue.

► **Lemma 7.** *Let  $A$  be any set-linearizable implementation of the set-concurrent queue with multiplicity. Then,*

1. All sequential executions of  $A$  are executions of the sequential queue.
2. All executions with no concurrent Dequeue operations are linearizable with respect to the sequential queue.
3. All executions with Dequeue operations returning distinct values are linearizable with respect to the sequential queue.
4. If two Dequeue operations return the same value in an execution, then they are concurrent.

## 13:12 Relaxed Queues and Stacks from Read/Write Operations

Following a similar approach to that in the previous section, Seq-Queue can be modified to obtain a Read/Write non-blocking implementation of a queue with multiplicity. The algorithm is modified as follows: (1) replace the Fetch&Inc object in Seq-Queue with a Read/Write wait-free Counter, (2) extend *Items* to a matrix to handle collisions, and (3) simulate the Swap operation with a Read followed by a Write. The correctness proof of the modified algorithm is similar to the correctness proof of Set-Conc-Stack. In fact, proving that the algorithm implements the set-concurrent pool with multiplicity is simple, the difficulty comes from the FIFO order requirement of the queue, which is shown through a simulation argument.

► **Theorem 8.** *There is a Read/Write non-blocking set-linearizable implementation of the queue with multiplicity.*

### 5 Implications

**Avoiding Costly Synchronization Operations/Patterns.** It is worth observing that Set-Conc-Stack and Set-Conc-Queue allow us to circumvent the linearization-related impossibility results in [12], where it is shown that every linearizable implementation of a queue or a stack, as well as other concurrent operation executions as encountered for example in work-stealing, must use either expensive Read-Modify-Write operations (e.g. Fetch&Inc and Compare&Swap) or Read-After-Write patterns [12] (i.e. a process writing in a shared variable and then reading another shared variable, may be performing operation on other variables in between).

In the simplest Read/Write Counter implementation we are aware of, the object is represented via a shared array  $M$  with an entry per process; process  $p_i$  performs Increment by incrementing its entry,  $M[i]$ , and Read by reading, one by one, the entries of  $M$  and returning the sum. Using this simple Counter implementation, we obtain from Set-Conc-Stack a set-concurrent stack implementation with multiplicity, devoided of (1) Read-Modify-Write operations, as only Read/Write operations are used, and (2) Read-After-Write patterns, as in both operations, Push and Pop, a process first reads and then writes. It similarly happens with Set-Conc-Queue.

**Work-stealing with multiplicity.** Our implementations also provide relaxed *work-stealing* solutions without expensive synchronization operation/patterns. Work-stealing is a popular technique to implement load balancing in a distributed manner, in which each process maintains its own *pool* of tasks and occasionally *steals* tasks from the pool of another process. In more detail, a process can Put and Take tasks in its own pool and Steal tasks from another pool. To improve performance, [30] introduced the notion of *idempotent work-stealing* which allows a task to be taken/stolen at least once instead of exactly once as in previous work. Using this relaxed notion, three different solutions are presented in that paper where the Put and Take operations avoid Read-Modify-Write operations and Read-After-Write patterns; however, the Steal operation still uses costly Compare&Swap operations.

Our set-concurrent queue and stack implementations provide idempotent work-stealing solutions in which no operation uses Read-Modify-Write operations and Read-After-Write patterns. Moreover, in our solutions both Take and Steal are implemented by Pop (or Dequeue), hence any process can invoke those operations, allowing more concurrency. If we insist that Take and Steal can be invoked only by the owner, *Items* can be a 1-dimensional array. Additionally, differently from [30], whose approach is practical, our queues and stacks with multiplicity are formally defined, with a clear and simple semantics.

**Out-of-order queues and stacks with multiplicity:** The notion of a *k-FIFO queue* is introduced in [28] (called *k-out-of-order queue* in [22]), in which items can be dequeued out of FIFO order up to an integer  $k \geq 0$ . More precisely, dequeuing the oldest item may require up to  $k + 1$  dequeue operations, which may return elements not younger than the  $k + 1$  oldest elements in the queue, or nothing even if the queue is not empty. [28] presents also a simple way to implement a *k-FIFO queue*, through  $p$  independent FIFO queue linearizable implementations. When a process wants to perform an operation, it first uses a *load balancer* to pick one of the  $p$  queues and then performs its operation. The value of  $k$  depends on  $p$  and the load balancer. Examples of load balancers are round-robin load balancing, which requires the use of Read-Modify-Write operations, and randomized load balancing, which does not require coordination but can be computationally locally expensive. As explained in [28], the notion of a *k-FIFO stack* can be defined and implemented similarly.

We can relax the *k-FIFO queues and stacks* to include multiplicity, namely, an item can be taken by several concurrent operations. Using  $p$  instances of our set-concurrent stack or queue Read/Write implementations, we can easily obtain set-concurrent implementations of *k-FIFO queues and stacks with multiplicity*, where the use of Read-Modify-Write operations or Read-After-Write patterns are in the load balancer.

## 6 Interval-Concurrent Queues with Weak-Emptiness Check

A natural question is if in Section 4 we could start with a wait-free linearizable queue implementation instead of Seq-Queue, which is only non-blocking, and hence derive a wait-free set-linearizable queue implementation with multiplicity. It turns out that it is an open question if there is a wait-free linearizable queue implementation from objects with consensus number two. (Concretely, such an algorithm would show that the queue belongs to the Common2 family of operations [4].) This question has been open for more than two decades [4] and there have been several papers proposing wait-free implementations of restricted queues [10, 18, 29, 16, 17], e.g., limiting the number of processes that can perform a type of operations.

```

Shared Variables:
  Tail : Fetch&Inc object initialized to 1
  Items[1, . . .] : Swap objects initialized to  $\perp$ 

Operation Enqueue( $x_i$ ) is
(01)  $tail_i \leftarrow Tail.Fetch\&Inc()$ 
(02)  $Items[tail_i].Write(x_i)$ 
(03) return true
end Enqueue

Operation Dequeue() is
(04)  $tail_i \leftarrow Tail.Read() - 1$ 
(05) for  $r_i \leftarrow 1$  up to  $tail_i$  do
(06)    $x_i \leftarrow Items[r_i].Swap(\perp)$ 
(07)   if  $x_i \neq \perp$  then return  $x_i$  end if
(08) end for
(09) return  $\epsilon$ 
end Dequeue

```

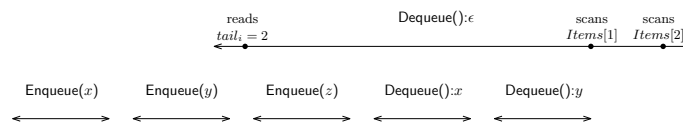
■ **Figure 6** A non-linearizable queue implementation (code for process  $p_i$ ).

**The Tail-Chasing Problem.** One of the main difficulties to solve when trying to design such an implementations using objects with consensus number two is that of reading the current position of the tail. This problem, which we call as *tail-chasing*, can be easily exemplified

## 13:14 Relaxed Queues and Stacks from Read/Write Operations

with the help of the *non-linearizable* queue implementation in Figure 6. The implementation is similar to *Seq-Stack* with the difference that *Dequeue* operations scan *Items* in the opposite order, i.e. from the head to the tail.

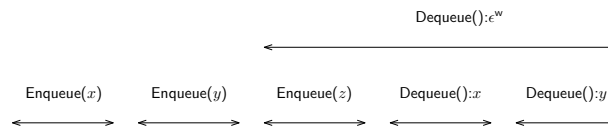
The problem with this implementation is that once a *Dequeue* has scanned unsuccessfully *Item* (i.e., the items that were in the queue were taken by “faster” operations), it returns  $\epsilon$ ; however, while the operation was scanning, more items could have been enqueued, and indeed it is not safe to return  $\epsilon$  as the queue might not be empty. Figure 7 describes an execution of the implementation that cannot be linearized because there is no moment in time during the execution of the *Dequeue* operation returning  $\epsilon$  in which the queue is empty. Certainly, this problem can be solved as in *Seq-Queue*: read the tail and scan again; thus, in order to complete, a *Dequeue* operation is forced to *chase* the current position of the tail until it is sure there are no new items.



■ **Figure 7** An example of the tail-chasing problem.

Inspired by this problem, below we introduce a relaxed interval-concurrent queue that allows a *Dequeue* operation to return a weak-empty value, with the meaning that the operation was not able take any of the items that were in the queue when it started but it was concurrent with all the *Dequeue* operation that took those items, i.e., it has a sort of *certificate* that the items were taken, and the queue might be empty. Then, we show that such a relaxed queue can be wait-free implemented from objects with consensus number two.

**A Wait-Free Interval-Concurrent Queue with Weak-Emptiness.** Roughly speaking, in our relaxed interval-concurrent queue, the state is a tuple  $(q, P)$ , where  $q$  denotes the state of the queue and  $P$  denotes the *pending* *Dequeue* operations that eventually return *weak-empty*, denoted  $\epsilon^w$ . More precisely,  $P[i] \neq \perp$  means that process  $p_i$  has a pending *Dequeue* operation.  $P[i]$  is a prefix of  $q$  and represents the remaining items that have to be dequeued so that the current *Dequeue* operation of  $p_i$  can return  $\epsilon^w$ . *Dequeue* operations taking items from the queue, also remove the items from  $P[i]$ , and the operation of  $p_i$  can return  $\epsilon^w$  only if  $P[i]$  is  $\epsilon$ . Intuitively, the semantics of  $\epsilon^w$  is that the queue *could* be empty as all items that were in the queue when the operations started have been taken. So this *Dequeue* operation virtually occurs after all the items have been dequeued.



■ **Figure 8** An interval-concurrent execution with a *Dequeue* operations returning weak-empty.

Figure 8 shows an example of an interval-concurrent execution of our relaxed queue where the *Dequeue* operation returning  $\epsilon^w$  is allowed to return only when  $x$  and  $y$  have been dequeued. Observe that this execution is an interval-linearization of the execution obtained from Figure 7 by replacing  $\epsilon$  with  $\epsilon^w$ .

► **Definition 9** (Interval-Concurrent Queue with Weak-Empty). *The universe of items that can be enqueued is  $\mathbf{N} = \{1, 2, \dots\}$  and the set of states is  $Q = \mathbf{N}^* \times (\mathbf{N}^* \cup \{\perp\})^n$ , with the initial state being  $(\epsilon, \perp, \dots, \perp)$ . Below, a subscript denotes the ID of the process invoking an operation. The transitions are the following:*

1. For  $(q, P) \in Q$ ,  $0 \leq t, \ell \leq n-1$ ,  $\delta(q, P, \text{Enqueue}(x), \text{Dequeue}_{i(1)}(), \dots, \text{Dequeue}_{i(t)}())$  contains the transition  $(q \cdot x, S, \langle \text{Enqueue}(x) : \text{true} \rangle, \langle \text{Dequeue}_{j(1)}() : \epsilon^w \rangle, \dots, \langle \text{Dequeue}_{j(\ell)}() : \epsilon^w \rangle)$ , satisfying that
  - a.  $i(k) \neq i(k')$ ,  $i(k) \neq$  the id of the process invoking  $\text{Enqueue}(x)$ , and  $j(k) \neq j(k')$ ,
  - b. for each  $i(k)$ ,  $P[i(k)] = \perp$ ,
  - c. for each  $j(k)$ , either  $P[j(k)] = \epsilon$ , or  $P[j(k)] = \perp$  and  $q = \epsilon$  and  $j(k) = i(k')$  for some  $k'$ ,
  - d. for each  $1 \leq s \leq n$ , if there is a  $k$  with  $s = j(k)$ , then  $S[s] = \perp$ ; otherwise, if there is  $k'$  with  $s = i(k')$ ,  $S[s] = q$ , else  $S[s] = P[s]$ .
2. For  $(x \cdot q, P) \in Q$ ,  $0 \leq t, \ell \leq n-1$ ,  $\delta(x \cdot q, P, \text{Dequeue}(), \text{Dequeue}_{i(1)}(), \dots, \text{Dequeue}_{i(t)}())$  contains the transition  $(q, S, \langle \text{Dequeue}() : x \rangle, \langle \text{Dequeue}_{j(1)}() : \epsilon^w \rangle, \dots, \langle \text{Dequeue}_{j(\ell)}() : \epsilon^w \rangle)$ , satisfying that
  - a.  $i(k) \neq i(k')$ ,  $i(k) \neq$  the id of the process invoking  $\text{Dequeue}()$ , and  $j(k) \neq j(k')$ ,
  - b. for each  $i(k)$ ,  $P[i(k)] = \perp$ ,
  - c. for each  $j(k)$ , either  $P[j(k)] = x$ , or  $P[j(k)] = \perp$  and  $q = \epsilon$  and  $j(k) = i(k')$  for some  $k'$ ,
  - d. for each  $1 \leq s \leq n$ , if there is a  $k$  with  $s = j(k)$ , then  $S[s] = \perp$ ; otherwise, if there is  $k'$  with  $s = i(k')$ ,  $S[s] = q$ , else  $S[s]$  is the string obtained by removing the first symbol of  $P[s]$  (which must be  $x$ ).
  - e. if  $x \cdot q = \epsilon$  and  $t, \ell = 0$ , then  $x \in \{\epsilon, \epsilon^w\}$ .

► **Remark 10.** Every execution of the interval-concurrent queue with no dequeue operation returning  $\epsilon^w$  is an execution of the sequential queue.

► **Lemma 11.** *Let  $A$  be any interval-linearizable implementation of the interval-concurrent queue with weak-empty. Then, (1) all sequential executions of  $A$  are executions of the sequential queue, and (2) all executions in which no Dequeue operation is concurrent with any other operation are linearizable with respect to the sequential queue.*

The algorithm in Figure 9, which we call **Int-Conc-Queue**, is an interval-linearizable wait-free implementation of a queue with weak-emptiness, which uses base objects with consensus number two. **Int-Conc-Queue** is a simple modification of **Seq-Queue** in which an **Enqueue** operation proceeds as in **Seq-Queue**, while a **Dequeue** operation scans *Items* at most two times to obtain an item, in both cases recording the number of taken items. If the two numbers are the same (cf. *double clean scan*), then the operations return  $\epsilon$ , otherwise it returns  $\epsilon^w$ .

► **Theorem 12.** *The algorithm Int-Conc-Queue (Figure 9) is a wait-free interval-linearizable implementation of the queue with weak-empty, using objects with consensus number two.*

**Interval-Concurrent Queue with Weak-emptiness and Multiplicity.** Using the techniques in Sections 3 and 4, we can obtain a Read/Write wait-free implementation of a even more relaxed interval-concurrent queue in which an item can be taken by several dequeue operations, i.e., with multiplicity. In more detail, the interval-concurrent queue with weak-emptiness is modified such that concurrent **Dequeue** operations can return the same item and are set-linearized in the same concurrency class, as in Definitions 1 and 5.

```

Shared Variables:
  Tail : Fetch&Inc object initialized to 1
  Items[1, ...] : Swap objects initialized to  $\perp$ 

Operation Enqueue( $x_i$ ) is
(01)  $tail_i \leftarrow Tail.Fetch\&Inc()$ 
(02)  $Items[tail_i].Write(x_i)$ 
(03) return true
end Enqueue

Operation Dequeue() is
(04) for  $k \leftarrow 1$  up to 2 do
(05)    $taken_i[k] \leftarrow 0$ 
(06)    $tail_i \leftarrow Tail.Read() - 1$ 
(07)   for  $r_i \leftarrow 1$  up to  $tail_i$  do
(08)      $x_i \leftarrow Items[r_i].Read()$ 
(09)     if  $x_i \neq \perp$  then
(10)        $x_i \leftarrow Items[r_i].Swap(\top)$ 
(11)       if  $x_i \neq \top$  then return  $x_i$  end if
(12)        $taken_i[k] \leftarrow taken_i[k] + 1$ 
(13)     end if
(14)   end for
(15) end for
(16) if  $taken_i[1] = taken_i[2]$  then return  $\epsilon$ 
(17)   else return  $\epsilon^w$ 
(18) end if
end Dequeue

```

■ **Figure 9** Wait-free interval-concurrent queue from consensus number 2 (code for  $p_i$ ).

We obtain a Read/Write wait-free interval-concurrent implementation of the queue with weak-emptiness and multiplicity by doing the following: (1) replace the Fetch&Inc object in Int-Conc-Queue with a Read/Write wait-free Counter, (2) extend *Items* to a matrix to handle collisions, and (3) simulate the Swap operation with a Read followed by a Write. Thus, we have:

► **Theorem 13.** *There is a Read/Write wait-free interval-linearizable implementation of the queue with weak-emptiness and multiplicity.*

## 7 Final Discussion

Considering classical data structures initially defined for sequential computing, this work has introduced new well-defined relaxations to adapt them to concurrency and investigated algorithms that implement them on top of “as weak as possible” base operations. It has first introduced the notion of set-concurrent queues and stacks with multiplicity, a relaxed version of queues and tasks in which an item can be dequeued more than once by concurrent operations. Non-blocking and wait-free set-linearizable implementations were presented, both based only on the simplest Read/Write operations. These are the first implementations of relaxed queues and stacks using only these operations. The implementations imply algorithms for idempotent work-stealing and out-of-order stacks and queues.

The paper also introduced a relaxed concurrent queue with weak-emptiness check, which allows a dequeue operation to return a “weak-empty certificate” reporting that the queue might be empty. A wait-free interval-linearizable implementation using objects with consensus number two was presented for such a relaxed queue. As there are only non-blocking linearizable (not relaxed) queue implementations using objects with consensus number two, it is an open question if there is such a wait-free implementation. The proposed queue relaxation allowed us to go from non-blocking to wait-freedom using only objects with consensus number two.

This work also can be seen as a work prolonging the results described in [14] where the notion of interval-linearizability was introduced and set-linearizability [33] is studied. It has shown that linearizability, set-linearizability and interval-linearizability constitute a hierarchy

of consistency conditions that allow us to formally express the behavior of non-trivial (and still meaningful) relaxed queues and stacks on top of simple base objects such as Read/Write registers. An interesting extension to this work is to explore if the proposed relaxations can lead to practical efficient implementations. Another interesting extension is to explore if set-concurrent or interval-concurrent relaxations of other concurrent data structures would allow implementations to be designed without requiring the stronger computational power provided by atomic Read-Modify-Write operations.

---

## References

- 1 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993. doi:10.1145/153724.153741.
- 2 Yehuda Afek, Eli Gafni, and Adam Morrison. Common2 extended to stacks and unbounded concurrency. *Distributed Comput.*, 20(4):239–252, 2007. doi:10.1007/s00446-007-0023-3.
- 3 Yehuda Afek, Guy Korland, and Eitan Yanovsky. Quasi-linearizability: Relaxed consistency for improved concurrency. In *Principles of Distributed Systems - 14th International Conference, OPODIS 2010, Tozeur, Tunisia, December 14-17, 2010. Proceedings*, pages 395–410, 2010. doi:10.1007/978-3-642-17653-1\_29.
- 4 Yehuda Afek, Eytan Weisberger, and Hanan Weisman. A completeness theorem for a class of synchronization objects (extended abstract). In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing, Ithaca, New York, USA, August 15-18, 1993*, pages 159–170, 1993. doi:10.1145/164051.164071.
- 5 Dan Alistarh, Trevor Brown, Justin Kopinsky, Jerry Zheng Li, and Giorgi Nadiradze. Distributionally linearizable data structures. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures, SPAA 2018, Vienna, Austria, July 16-18, 2018*, pages 133–142, 2018. doi:10.1145/3210377.3210411.
- 6 Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. The spraylist: a scalable relaxed priority queue. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2015, San Francisco, CA, USA, February 7-11, 2015*, pages 11–20, 2015. doi:10.1145/2688500.2688523.
- 7 James Aspnes, Hagit Attiya, and Keren Censor-Hillel. Polylogarithmic concurrent data structures from monotone circuits. *J. ACM*, 59(1):2:1–2:24, 2012. doi:10.1145/2108242.2108244.
- 8 James Aspnes, Hagit Attiya, Keren Censor-Hillel, and Faith Ellen. Limited-use atomic snapshots with polylogarithmic step complexity. *J. ACM*, 62(1):3:1–3:22, 2015. doi:10.1145/2732263.
- 9 Hagit Attiya, Amotz Bar-Noy, Danny Dolev, David Peleg, and Rüdiger Reischuk. Renaming in an asynchronous environment. *J. ACM*, 37(3):524–548, 1990. doi:10.1145/79147.79158.
- 10 Hagit Attiya, Armando Castañeda, and Danny Hendler. Nontrivial and universal helping for wait-free queues and stacks. *J. Parallel Distributed Comput.*, 121:1–14, 2018. doi:10.1016/j.jpdc.2018.06.004.
- 11 Hagit Attiya, Rachid Guerraoui, Danny Hendler, and Petr Kuznetsov. The complexity of obstruction-free implementations. *J. ACM*, 56(4):24:1–24:33, 2009. doi:10.1145/1538902.1538908.
- 12 Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin T. Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 487–498, 2011. doi:10.1145/1926385.1926442.
- 13 Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. The renaming problem in shared memory systems: An introduction. *Comput. Sci. Rev.*, 5(3):229–251, 2011. doi:10.1016/j.cosrev.2011.04.001.



## 13:18 Relaxed Queues and Stacks from Read/Write Operations

- 14 Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. Unifying concurrent objects and distributed tasks: Interval-linearizability. *J. ACM*, 65(6):45:1–45:42, 2018. doi:10.1145/3266457.
- 15 Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. What can be done with consensus number one: Relaxed queues and stacks. *CoRR*, abs/2005.05427, 2020. arXiv:2005.05427.
- 16 Matei David. A single-enqueuer wait-free queue implementation. In *Distributed Computing, 18th International Conference, DISC 2004, Amsterdam, The Netherlands, October 4-7, 2004, Proceedings*, pages 132–143, 2004. doi:10.1007/978-3-540-30186-8\_10.
- 17 Matei David, Alex Brodsky, and Faith Ellen Fich. Restricted stack implementations. In *Distributed Computing, 19th International Conference, DISC 2005, Cracow, Poland, September 26-29, 2005, Proceedings*, pages 137–151, 2005. doi:10.1007/11561927\_12.
- 18 David Eisenstat. A two-enqueuer queue. *CoRR*, abs/0805.0444, 2008. arXiv:0805.0444.
- 19 Faith Ellen, Danny Hendler, and Nir Shavit. On the inherent sequentiality of concurrent objects. *SIAM J. Comput.*, 41(3):519–536, 2012. doi:10.1137/08072646X.
- 20 Andreas Haas, Thomas A. Henzinger, Andreas Holzer, Christoph M. Kirsch, Michael Lippautz, Hannes Payer, Ali Sezgin, Ana Sokolova, and Helmut Veith. Local linearizability for concurrent container-type data structures. In *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada*, pages 6:1–6:15, 2016. doi:10.4230/LIPIcs.CONCUR.2016.6.
- 21 Andreas Haas, Michael Lippautz, Thomas A. Henzinger, Hannes Payer, Ana Sokolova, Christoph M. Kirsch, and Ali Sezgin. Distributed queues in shared memory: multicore performance and scalability through quantitative relaxation. In *Computing Frontiers Conference, CF'13, Ischia, Italy, May 14 - 16, 2013*, pages 17:1–17:9, 2013. doi:10.1145/2482767.2482789.
- 22 Thomas A. Henzinger, Christoph M. Kirsch, Hannes Payer, Ali Sezgin, and Ana Sokolova. Quantitative relaxation of concurrent data structures. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 317–328, 2013. doi:10.1145/2429069.2429109.
- 23 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991. doi:10.1145/114005.102808.
- 24 Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.
- 25 Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. doi:10.1145/78969.78972.
- 26 Damien Imbs and Michel Raynal. Help when needed, but no more: Efficient read/write partial snapshot. *J. Parallel Distributed Comput.*, 72(1):1–12, 2012. doi:10.1016/j.jpdc.2011.08.005.
- 27 Christoph M. Kirsch, Michael Lippautz, and Hannes Payer. Fast and scalable, lock-free k-fifo queues. In *Parallel Computing Technologies - 12th International Conference, PaCT 2013, St. Petersburg, Russia, September 30 - October 4, 2013. Proceedings*, pages 208–223, 2013. doi:10.1007/978-3-642-39958-9\_18.
- 28 Christoph M. Kirsch, Hannes Payer, Harald Röck, and Ana Sokolova. Performance, scalability, and semantics of concurrent FIFO queues. In *Algorithms and Architectures for Parallel Processing - 12th International Conference, ICA3PP 2012, Fukuoka, Japan, September 4-7, 2012, Proceedings, Part I*, pages 273–287, 2012. doi:10.1007/978-3-642-33078-0\_20.
- 29 Zongpeng Li. Non-blocking implementations of queues in asynchronous distributed shared-memory systems. Master's thesis, University of Toronto, 2001. URL: <http://hdl.handle.net/1807/16583>.
- 30 Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat. Idempotent work stealing. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2009, Raleigh, NC, USA, February 14-18, 2009*, pages 45–54, 2009. doi:10.1145/1504176.1504186.

- 31 Mark Moir and James H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Sci. Comput. Program.*, 25(1):1–39, 1995. doi:10.1016/0167-6423(95)00009-H.
- 32 Mark Moir and Nir Shavit. Concurrent data structures. In *Handbook of Data Structures and Applications*. Chapman and Hall/CRC, 2004. doi:10.1201/9781420035179.ch47.
- 33 Gil Neiger. Set-linearizability. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing, Los Angeles, California, USA, August 14-17, 1994*, page 396, 1994. doi:10.1145/197917.198176.
- 34 Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 456–471, 2013. doi:10.1145/2517349.2522739.
- 35 Hannes Payer, Harald Röck, Christoph M. Kirsch, and Ana Sokolova. Scalability versus semantics of concurrent FIFO queues. In *Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing, PODC 2011, San Jose, CA, USA, June 6-8, 2011*, pages 331–332, 2011. doi:10.1145/1993806.1993869.
- 36 Michel Raynal. *Concurrent Programming - Algorithms, Principles, and Foundations*. Springer, 2013. doi:10.1007/978-3-642-32027-9.
- 37 Hamza Rihani, Peter Sanders, and Roman Dementiev. Brief announcement: Multiqueues: Simple relaxed concurrent priority queues. In *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2015, Portland, OR, USA, June 13-15, 2015*, pages 80–82, 2015. doi:10.1145/2755573.2755616.
- 38 Nir Shavit. Data structures in the multicore age. *Commun. ACM*, 54(3):76–84, 2011. doi:10.1145/1897852.1897873.
- 39 Nir Shavit and Gadi Taubenfeld. The computability of relaxed data structures: queues and stacks as examples. *Distributed Comput.*, 29(5):395–407, 2016. doi:10.1007/s00446-016-0272-0.
- 40 Edward Talmage and Jennifer L. Welch. Relaxed data types as consistency conditions. *Algorithms*, 11(5):61, 2018. doi:10.3390/a11050061.
- 41 Gadi Taubenfeld. *Synchronization algorithms and concurrent programming*. Prentice Hall, 2006.
- 42 Tingzhe Zhou, Maged M. Michael, and Michael F. Spear. A practical, scalable, relaxed priority queue. In *Proceedings of the 48th International Conference on Parallel Processing, ICPP 2019, Kyoto, Japan, August 05-08, 2019*, pages 57:1–57:10, 2019. doi:10.1145/3337821.3337911.



# Fast and Space-Efficient Queues via Relaxation

Dempsey Wade

Bucknell University, Lewisburg, PA, USA

Edward Talmage<sup>1</sup>

Computer Science Department, Bucknell University, Lewisburg, PA, USA

edward.talmage@bucknell.edu

---

## Abstract

Efficient message-passing implementations of shared data types are a vital component of practical distributed systems, enabling them to work on shared data in predictable ways, but there is a long history of results showing that many of the most useful types of access to shared data are necessarily slow. A variety of approaches attempt to circumvent these bounds, notably weakening consistency guarantees and relaxing the sequential specification of the provided data type. These trade behavioral guarantees for performance. We focus on relaxing the sequential specification of a first-in, first-out queue type, which has been shown to allow faster linearizable implementations than are possible for traditional FIFO queues without relaxation.

The algorithms which showed these improvements in operation time tracked a complete execution history, storing complete object state at all  $n$  processes in the system, leading to  $n$  copies of every stored data element. In this paper, we consider the question of reducing the space complexity of linearizable implementations of shared data types, which provide intuitive behavior through strong consistency guarantees. We improve the existing algorithm for a relaxed queue, showing that it is possible to store only one copy of each element in a shared queue, while still having a low amortized time cost. This is one of several important steps towards making these data types practical in real world systems.

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

**Keywords and phrases** Shared Data Structures, Message Passing, Relaxed Data Types, Space Complexity

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2020.14

**Acknowledgements** We would like to thank Anh Kieu, Shane Staret, and Jimmy Wei for helping find references.

## 1 Introduction & Related Work

Because they present the same interface as sequential data types, shared memory objects are a relatively intuitive way to program access to shared data by many processors. Unfortunately, in a distributed computation setting, physical shared memory is usually not possible and processes communicate by sending messages. Programming in a message passing system is more difficult, since there tend to be many messages in transit at once, on many communication links, and their causal and temporal relationships may be masked by variable delays. To hide this difficulty and make distributed programming easier and less error-prone, there is much work on implementing shared memory objects as an abstraction layer on top of message passing systems. As more and more computing moves to distributed and cloud systems, the ability to write programs that interact with shared data in predictable and efficient ways continues to grow, so it is vital that we work to provide the best shared data structure implementations possible.

---

<sup>1</sup> Corresponding author



Existing work on shared data types has given implementations for both specific and arbitrary data types, but it has also shown that all operations with certain often-desirable properties are inherently slow, requiring a delay proportional to the maximum time a message may take in transit to ensure knowledge of preceding and concurrent operation invocations [8, 11, 6]. In a widely distributed system, such a delay could easily be on the order of hundreds of milliseconds, which is more than enough to negatively impact a human user's experience, and is extremely costly to a computation using such a shared data object.

There are several approaches to circumvent this lower bound, some of which the community has explored for decades and some of which are newer. The lower bounds mentioned before generally apply to *linearizable* implementations, in which it is possible to reduce a concurrent execution to an equivalent sequential one without reordering non-concurrent operations. Weaker consistency conditions, such as *sequential consistency* or *eventual consistency*, do not have the same lower bounds, but they also do not provide the same guarantees [2, 10] or intuitive correspondence to sequential structures. Eventual consistency is widely used in commercial applications but gives very weak guarantees, making it difficult to reason about the expected behavior of interactions with shared data. Even for stronger conditions like sequential consistency, the practical effects of weakened guarantees can be hard to anticipate and seem counter-intuitive, making them less attractive in practice.

Another approach called *relaxation*, developed in [1] and formalized in [5], allows better performance in a linearizable system [5, 9]. By weakening the guarantees of the data type's sequential specification, more possible responses to a particular operation are allowed, and this limited non-determinism can be exploited to eliminate the need for processes to synchronize in every operation instance. Instead, updates can be sent in the background, allowing quick responses and high throughput. Occasional synchronization is necessary, keeping the worst-case time complexity high, but relaxed data types can have a much lower amortized cost per operation than is possible for unrelaxed types.

While [9] proved the possibility of these performance gains, it did so in a theoretical model without many of the difficulties present in real systems. Assumptions of known message delays, free storage, and always-correct processes do not translate well to practical implementations. In this paper, we seek to take a first step towards removing these assumptions by reducing the space required to implement relaxed queues, while still maintaining good time performance. Future work is still necessary to remove other idealized model assumptions and build practical implementations of these types.

The performance gains possible with relaxed data structures, particularly queues, come at the cost of weakened guarantees on the order of data retrieval. For example, the relaxation we primarily consider in this work merely guarantees that some old element in the structure, not necessarily the single oldest, is returned by each *Dequeue* instance. Such a weakening reduces the usefulness of the data type in many cases, since we can no longer be confident which element we get when we retrieve one from the structure. But because the type still provides some ordering, these objects are still of use in applications where response time is more important than exact ordering. For example, consider a distributed job queue, where the primary goal is to execute a large computation as quickly as possible. While we intuitively want to send tasks in exact FIFO order, if they are being completed concurrently, their completion order may not exactly match their start order, so relaxing the start order will not adversely affect the computation. Similarly, online shopping applications, such as for high-demand, limited-run items like concert tickets, demand very quick response time, or customers are left frustrated while they wait to find out whether they were among the first to request a product. By relaxing the order of customers in some cases, the average wait

time may decrease, without any customer waiting longer than they would in an unrelaxed system. Care is necessary to avoid disadvantaging customers who made their request earlier, but this is possible by counting requests. As a tangible example, consider the problem of selling 1000 identical tickets for lawn seating at a concert. The exact order in which requests are processed has no bearing on correctness, as long as the first 1000 customers are those who get the tickets. Since we do not relax *Enqueue* (which already has good performance), we know that we store requests in the order they arrive, and merely need to mark which are the first 1000. With a fast relaxed *Dequeue*, we can then process all the requests more quickly, leading to happier customers and lower load on the ticket servers.

## Our Contribution

The algorithm for arbitrary data types in [11], on which the algorithm in [9] for relaxed types is based, keeps a complete copy of the shared data locally on every process and updates these copies based on operations invoked throughout the system. This is highly inefficient, especially for data types like Queues, Stacks, and Heaps in which reading an element also removes it, so only a single process will ever need the value of each stored data object. The space overhead of full replication was necessary for those algorithms to achieve their low time complexity, as they avoided waiting for round-trip messages by having every process simulate the shared object by executing all operation instances on its local copy of the structure.

We consider only linearizable implementations of data types, since they provide the strongest, most intuitive restrictions on concurrent behavior. The idea of only partially replicating data elements has been more thoroughly explored in the context of causal consistency [4, 12, 3] where, despite the weaker consistency condition it is difficult or impossible to store fewer than  $n$  copies of the data and maintain consistency.

We show that by exploiting the same properties of relaxation that allow a structure to have lower time cost, we can also reduce storage to only a single copy of each data element in the system. This gives a reduction in space complexity by a factor of  $n$  over the existing work. We here present this solution for one particular relaxation of FIFO queues as a proof of concept. For this relaxation, we increase the amortized time complexity of the costly *Dequeue* operation by approximately a factor of 2 over the relaxed queue implementation in [9], but with reasonable levels of relaxation still achieve amortized time below the lower bound for unrelaxed queues. We also show that this is better than is possible in an unrelaxed queue implementation. In the future, we intend to explore other relaxations, where we expect to match the time complexity of the best-known algorithm while still reducing the total space complexity by a factor of  $n$ .

Our solution is still somewhat idealized, as we keep assumptions about known message delay bounds and correct processes. We are working separately on fault-tolerant implementations, with the aim of eventually combining improvements along different dimensions. In fact, a real-world solution will probably not want to reduce space complexity quite as far as we do here, since some replication is necessary to prevent data loss in the presence of faults. However, we feel it worthwhile to explore the bounds of possible space savings and the tradeoff of space versus time on their own merits. This helps demonstrate the essential parts of efficient implementations and educates our ongoing work to build practically useful structures.

## 2 Model & Definitions

We consider sequential data type specifications consisting of two parts: a list of operations, with argument and return types, and a list of legal sequences of invocation-response pairs of those operations. In the sequential setting, an operation's invocation must be immediately followed by its response. In a concurrent setting, the argument and response of an operation may occur at different times. An operation instance is an invocation of an operation, which specifies an argument, together with a corresponding response. We require that the set of legal sequences in a data type specification be prefix-closed and complete, meaning that any prefix of a legal sequence is legal and after any sequence  $\rho$ , for any invocation  $i$  there must be a response  $r$  forming an operation instance  $(i, r)$  such that  $\rho \cdot (i, r)$  is legal.

We focus on the queue data type, since its ordering properties lend themselves to intuitive relaxations. Specifically, in this paper we implement queues with Out-of-Order  $k$ -relaxed *Dequeue*. Informally, this is a FIFO queue in which each *Dequeue*, instead of being required to return and remove the oldest element in the queue, may return and remove any of the  $k$  oldest elements. For analysis, we will also refer to the derived parameter  $\ell := \lfloor k/n \rfloor$ .

► **Definition 1.** *A queue with Out-of-Order  $k$ -relaxed *Dequeue* provides two operations:*

1. *Enqueue( $x, -$ ) takes one value  $x$  as its argument and returns nothing.*
2. *Dequeue( $-, r$ ) takes no argument and returns one value  $r$ .*

*Let  $\perp$  be a special symbol to indicate an empty queue. The empty sequence is legal and, if  $\rho$  is a legal sequence,*

- *$\rho \cdot \text{Enqueue}(x, -)$  is legal for any  $x \neq \perp$  which is not the argument of an *Enqueue* in  $\rho$ .<sup>2</sup>*
- *$\rho \cdot \text{Dequeue}(-, r), r \neq \perp$ , is legal if  $r$  is the argument of one of the first  $k$  *Enqueue*( $y$ ) instances in  $\rho$  s.t. *Dequeue*( $-, y$ ) is not in  $\rho$ .*
- *$\rho \cdot \text{Dequeue}(-, \perp)$  is legal if there are fewer than  $k$  *Enqueue*( $y$ ) instances in  $\rho$  s.t. *Dequeue*( $-, y$ ) is not in  $\rho$ .*

We adopt the model of [9]: We consider a system of  $n$  processes which can communicate by sending point-to-point messages to each other. This is a partially-synchronous model, where each process has a local clock running at the same rate as real time, but with an unknown offset, and processes know that every message takes between  $d - u$  and  $d$  real time in transit. We assume that local computation is instantaneous to focus on the communication costs which arise in the algorithm. Each process interacts with a user by allowing them to invoke operations and by providing return values to those invocations. We thus model each process with a state machine whose transitions are triggered by three types of events: message arrival, timer expiration, and operation invocation, and which can set timers, send messages, and/or generate operation responses in each step.

A schedule for each process describes the sequence of states and transitions of its state machine. A run of an algorithm consists of a schedule for each process, where each transition has an associated real time. A run is admissible if the times associated with each process' transitions are monotonically non-decreasing and interaction with each process starts with an operation invocation and then alternates responses and invocations. This prevents a user from invoking an operation until its previous invocation has finished. A run is complete if every message sent is received and each process' schedule is infinite or ends with no timers set. Note that this assumes that all processes are correct and do not crash.

---

<sup>2</sup> We assume that arguments to *Enqueue* are unique. This can be achieved by another abstraction layer adding tags such as timestamps to elements.



We assume that local clocks have previously been synchronized by an algorithm such as that of [7], which yields an optimal bound of  $\epsilon \leq (1 - 1/n)u$  on clock skew, the difference between any two local clocks.

We consider algorithms implementing data type specifications in this message passing model which satisfy a liveness condition, that every operation invocation has a matching response and vice versa, and linearizability, which says that for every complete, admissible run of the algorithm, there is a permutation, called a linearization, of the operation instances in the run which is legal by the data type specification and respects the real-time order of instances which do not overlap in real time. We require algorithms to be eventually quiescent, which means that if users stop invoking operations, every process' schedule will be finite—processes eventually stop setting timers and sending messages.

The time complexity of an operation  $OP$  in this model, denoted  $|OP|$ , is the maximum over all instances in all complete, admissible runs of the real time between the invocation and response of a single instance of that operation. We are also interested in the amortized time complexity of  $OP$ , which is the maximum over all complete, admissible runs of the average real time between invocation and response of every instance of  $OP$ . We assume that local computation is instantaneous, partly because it is practically much faster than communication time and partly because we are focused on minimizing the cost of communication-related delays.

To measure space complexity of our queue implementations, we introduce the parameter  $T$ , which represents the maximum number of data elements concurrently in the queue. That is, in a sequence  $\pi$  of operation instances,  $T$  is the maximum over all prefixes  $\rho$  of  $\pi$  of the number of *Enqueue* instances in  $\rho$  minus the number of *Dequeue* instances in  $\rho$  which return a non- $\perp$  value. To focus on the principles of shared data objects, we only measure the amount of data stored, not local variables used for the algorithm or buffers holding unprocessed messages.

### 3 Lower Bound on Unrelaxed Queues

We begin with a brief argument for the worst-case time complexity of any algorithm for an unrelaxed queue which stores only one copy of each data element. Our algorithm will match this space complexity and worst-case time bound, which shows that our algorithm is not a step backward from an unrelaxed queue with the same space complexity.

► **Theorem 2.** *Any algorithm linearizably implementing an unrelaxed queue which stores only one copy of each element must have  $|Dequeue| \geq 2d$ .*

**Proof.** Suppose that some algorithm  $A$  linearizably implements a queue and only stores one copy of each element. Consider a run in which one process enqueues several elements, then nothing happens until the system is quiescent. Since  $A$  stores only one copy of each element, the oldest element *head* is stored at a single process, which we'll call  $p_h$ . Suppose that some other process  $p_d$  then invokes a *Dequeue*. Since there are no concurrent operation instances,  $p_d$  must return *head*. But  $p_d$  doesn't know what *head* is, so must retrieve it from  $p_h$ . Since the system was in a quiescent state,  $p_h$  must wait to hear from  $p_d$  before sending *head*, and the upper bound on message delay implies that  $p_d$  will not have *head* available to return until up to  $2d$  after invocation. ◀

In a system that does not satisfy eventual quiescence, we could prove the same result by showing that if *head* is in transit, a process that is not the recipient of a message carrying *head* can invoke a *Dequeue* and it will still need to wait up to  $2d$  time before receiving *head*.

The exception, and why we restrict ourselves to the domain of eventually quiescent algorithms, are algorithms that effectively use the message channels for memory. By broadcasting all elements immediately upon reception, instead of storing them, such an algorithm could prevent a *Dequeue* instance from needing a long delay. This would lead to an unconscionably large message complexity, even in the absence of activity on the data structure.

## 4 Algorithm

Our algorithm is a modification of that in [9], as we want to maintain its improvements in time complexity. That algorithm uses a system of timers to ensure that all processes execute all invocations on their local copies of the queue in the same order. This, coupled with deterministic execution, ensures that all processes maintain the same state and can provide consistent and correct return values. To enable the majority of *Dequeue* instances to return after only local computation, the algorithm used an element-claiming system to divide the  $k$  values which were possibilities for a legal *Dequeue* return value among the  $n$  processes. As long as a process had a claimed element when a *Dequeue* invocation arrived, the algorithm responded quickly to the user and coordinated with other processes in the background. When a process ran out of claimed elements, the next *Dequeue* invocation was forced to wait until the process was sure its local copy of the queue was up to date to claim ownership of more elements and generate a *Dequeue* response.

We want to avoid having a complete copy of the shared state at every process, so we add mechanisms for determining which process stores each element. This is a two-stage system, with each element initially stored at one process, but then moved to a (potentially) different process which claims it. This transfer is necessary for a process to be able to return its claimed elements without waiting for communication with other processes, enabling the common case of most *Dequeue* instances returning without waiting for communication.

### 4.1 Description

We first give an intuitive description of our algorithm's behavior. The algorithm is event-driven, where possible events are operation invocation, message arrival, and timer expiration. Recall that we assume local computation is instantaneous, so events cannot interrupt other event handlers.<sup>3</sup>

When a user invokes an operation at a particular process  $p_i$ , the appropriate handler (lines 1-4 for *Enqueue*, 5-11 for *Dequeue*) will announce the invocation to every process, including  $p_i$ . When each process receives such an announcement of an invocation  $op$ , the message handler in lines 15-20 sets a timer to wait  $u + \epsilon$  time ( $u$  to account for variation in message delay and  $\epsilon$  to correct for clock skew) ensuring that it receives all invocations with smaller timestamps. The process will then locally execute, in increasing timestamp order, all invocations with smaller timestamps, ending with  $op$ , via the while loop in lines 21-25. In [9], this guaranteed that all processes follow the same sequence of local operation executions, allowing them to keep their local views of the shared queue synchronized. In our algorithm, we do not store the full state of the shared queue at each process, so cannot make as strong a claim. Instead, we track the number of each type of operation (modulo  $n$ ) and current size of the simulated queue, which enables us to determine which process should store and retrieve the data elements involved in each *Enqueue* or *Dequeue* instance, respectively.

---

<sup>3</sup> We name functions in the pseudocode based on how they may be called: **HandleEvents** respond to external events, while **Functions** are called internally.

We can respond to certain operation invocations before their execution is complete at all processes, as soon as we know the correct return value. The algorithm will propagate the instance's effects in the background to ensure correct state. All *Enqueue* instances can thus return quickly, since they have no return value. Similarly, *Dequeue* instances can quickly return one of a process' claimed elements, if there are any. Lines 12-14 generate these fast responses. When a process runs out of claimed elements, however, quick returns to *Dequeue* invocations are not possible, dividing *Dequeue* instances into two types, fast and slow. For slow *Dequeues*, the invoking process must claim new elements, which requires waiting until it knows about all preceding operation instances so that all processes agree which elements it claims. This occurs when processes locally execute the slow *Dequeue* instance, which sends the newly-claimed element to the invoking process as a *restock* message. Once the invoking process receives a *restock*, it knows it has an element that is a correct return value for the *Dequeue* instance and will not be returned by any other process, so the slow *Dequeue* can return to the user. The algorithm does this in lines 39-53 by a similar logical structure as that which ensures local execution of all instances in timestamp order. Here, we ensure that *restock* elements are claimed, or returned by slow *Dequeue* instances, in the correct order.

There are two primary improvements in this algorithm over that of [9]. First, and central to this paper's result, we note that when processes receive an announcement of a new *Enqueue* instance, they do not all need to store a copy of the argument. Instead, we separate *stored* elements from those which processes have *claimed*. Only one process saves the new element, putting it in a local *stored* queue. When a process claims that element, the storing process can send and delete it, since it will be saved in the claiming process' *claimed* queue.

Using the number of *Enqueue* instances which have happened so far, the algorithm distributes enqueued elements in a round-robin fashion to achieve balanced storage. When processes claim elements and remove them from storage, we similarly remove them in round-robin order using the saved number of *Dequeue* instances, which guarantees a FIFO ordering of *stored* elements across all processes. Note that this order does not hold for *Dequeue* return values, since a process can claim elements, then sit idle while other processes remove elements added to the queue since its claimed elements.

Our second improvement is the restocking procedure: when a process invokes a *Dequeue*, its announcement of that invocation also serves as a request to claim a new element. If the invoking process has no claimed elements, it must wait for the new element to arrive from the process storing it. If the invoking process has claimed elements, restocking occurs in the background, with the effect that if *Dequeue* invocations are not too frequent at any one process, all *Dequeue* instances in a run could be fast. This restocking system increases the worst-case time of a *Dequeue* to approximately  $2d$ , where the original algorithm had a worst-case time of approximately  $d$ , but this tradeoff is necessary to reduce our storage requirements. As detailed later in the paper, we still have a lower amortized time complexity than is possible without relaxation.

Pseudocode for our relaxed queue is in Algorithms 1 and 2. It uses local FIFO queues *claimed* and *stored* and min-priority queues *Pending* and *Restocks*, which are keyed on the timestamps (lexicographically-ordered pairs containing local clock values and process ids) of the instances they store.

## 4.2 Correctness

To prove our algorithm is a correct, linearizable implementation of a queue with Out-of-Order  $k$  relaxed *Dequeue*, we will show that every invocation has a response, then construct a linearization of those instances based on the timestamps assigned when they are invoked, and show that every return value is legal by the data type specification.

■ **Algorithm 1** Pseudocode for each  $p_i$  implementing a Queue with Out-of-Order  $k$ -relaxed *Dequeue*.

---

```

1: HandleEvent ENQUEUE( $val$ )
2:    $ts = \langle localtime, i \rangle$ 
3:   send ( $enq, val, ts$ ) to all
4:    $setTimer(\epsilon, \langle enq, val, ts \rangle, respond)$ 
5: HandleEvent DEQUEUE
6:    $ts = \langle localtime, i \rangle$ 
7:    $val = claimed.dequeue()$ 
8:   if  $val \neq \perp$  then
9:     send ( $fastDeq, val, ts$ ) to all
10:     $setTimer(\epsilon, \langle fastDeq, val, ts \rangle, respond)$ 
11:   else send ( $slowDeq, \perp, ts$ ) to all
12: HandleEvent EXPIRETIMER( $\langle op, val, ts \rangle, respond$ )
13:   if  $op == fastDeq$  then Generate Dequeue response with return value  $val$ 
14:   else Generate Enqueue response with no return value
15: HandleEvent RECEIVE ( $op, val, ts$ ) FROM  $p_j$ 
16:   if  $op \in \{fastDeq, slowDeq\}$  then
17:      $Restocks.insert(\langle op, val, ts \rangle)$ 
18:      $setTimer(d + 2u + \epsilon, \langle op, val, ts \rangle, restock)$ 
19:      $Pending.insert(\langle op, val, ts \rangle)$ 
20:      $setTimer(u + \epsilon, \langle op, val, ts \rangle, execute)$ 
21: HandleEvent EXPIRETIMER( $\langle op, val, ts \rangle, execute$ )
22:   while  $ts \geq Pending.min()$  do
23:      $\langle op', val', ts' \rangle = Pending.extractMin()$ 
24:      $executeLocally(op', val', ts')$ 
25:      $cancelTimer(\langle op', arg', \langle t, j \rangle \rangle, execute)$ 
26: Function EXECUTELOCALLY( $op, val, ts$ )
27:   if  $op == enq$  then
28:     if  $enqueueCount == i$  then
29:       if  $clean$  and  $size < k$  then
30:          $claimed.enqueue(val)$ 
31:       else  $stored.enqueue(val)$ 
32:        $enqueueCount += 1 \pmod n$ 
33:        $size += 1$ 
34:   else
35:     if  $restockCount == i$  then send ( $restock, stored.dequeue(), \langle op, val, ts \rangle$ ) to  $p_j$ 
36:      $restockCount += 1 \pmod n$ 
37:      $size -= 1$ 
38:      $clean = (size == 0)$ 
39: HandleEvent RECEIVE ( $restock, restockVal, \langle op, val, \langle t, i \rangle \rangle$ ) FROM  $p_j$ 
40:    $Restocks.update(\langle op, val, \langle t, i \rangle \rangle, restockVal)$ 
41: Function EXPIRETIMER( $\langle op, val, ts, restockVal \rangle, restock$ )
42:   while  $ts \geq Restocks.min()$  do
43:      $\langle op', val', ts', restockVal' \rangle = Restocks.extractMin()$ 
44:      $EXECUTERESTOCK(op', val', ts', restockVal')$ 
45:      $cancelTimer(\langle op', arg', ts', restockVal' \rangle, restock)$ 

```

---

■ **Algorithm 2** Algorithm 1, continued.

---

```

46: Function EXECUTERESTOCK( $op, val, \langle *, j \rangle, restockVal$ )
47:   if  $op == fastDeq$  and  $j == i$  then
48:      $claimed.Enqueue(restockVal)$ 
49:   if  $op == slowDeq$  and  $j == i$  then
50:      $returnVal = claimed.Dequeue()$ 
51:     if  $returnVal == \perp$  then  $returnVal == restockVal$ 
52:     else  $claimed.Enqueue(restockVal)$ 
53:   Generate  $Dequeue$  response with return value  $returnVal$ 

```

---

Let  $R$  be an arbitrary complete, admissible run of the algorithm. We assume that if multiple events happen at the same process at exactly the same real time, message receptions occur before timer expirations, but events of the same type may occur in any order. An operation invocation's *timestamp* is the value of the variable  $ts$  defined in line 2 or 6 for  $Enqueue$  or  $Dequeue$  invocations, respectively.

We omit the proofs for Lemmas 3 and 4 for the sake of space, since they are fundamentally the same as proofs in [9].

► **Lemma 3.** *Each operation invocation in  $R$  causes exactly one response.*

This defines the set of operation instances in  $R$ , by pairing each invocation with the resultant response. We say that an operation instance's timestamp is that of its invocation.

► **Lemma 4.** *Every process locally executes every operation instance exactly once, in timestamp order.*

► **Construction 1.** Let  $\pi$  be the sequence of all operations instance in  $R$ , sorted by timestamp order.

► **Lemma 5.**  $\pi$  respects the order of non-overlapping operation instances in  $R$ .

**Proof.** Suppose in contradiction that  $op_2$  responds before  $op_1$ 's invocation, but  $ts(op_1) < ts(op_2)$ , so  $op_1$  precedes  $op_2$  in  $\pi$ . Every operation instance takes at least  $\epsilon$  time to respond, by the timers in lines 4 and 10. Thus,  $op_2$ 's invocation must be at least  $\epsilon$  real time before  $op_1$ 's. But local clocks are skewed by at most  $\epsilon$ , so  $ts(op_2)$  must be less than or equal to  $ts(op_1)$ , contradicting our assumption and proving the claim. ◀

► **Lemma 6.** *At any time, there are no more than  $k$  elements in the union of all processes' claimed queues and the set of restock messages in transit.*

**Proof.** We observe that there are only two ways that elements can be added to a *claimed* queue. First, in a *clean* state, which means that there have been no *Dequeues* since the queue was last empty, *Enqueue* instances can add their arguments directly to *claimed* queues in line 30. This cannot cause there to be more than  $k$  elements in all processes' *claimed* queues, by the check in line 29.

Second, we add elements to *claimed* when restocking after a *Dequeue* instance, in lines 48 and 52. Elements are only added to *claimed* after removing an element from that process' *claimed*, either in line 7 or line 50, no *Dequeue* instance can increase the size of any process' *claimed* queue above the maximum size of that queue set by *Enqueue* instances.

The only time an element is sent in a *restock* message is after a *Dequeue* instance. If that instance was fast, then it removed a *claimed* element, so sending the *restock* message does not increase the number of *claimed* or *restocking* elements. If the *Dequeue* instance

## 14:10 Space-Efficient Relaxed Queues

was slow, then the invoking process had no *claimed* elements. Sending the *restock* message could increase the total number of elements claimed or in transit, but since the invoking process' *claimed* queue was empty,  $k \geq n$ , and *Enqueue* instances which add to *claimed* do so in round-robin fashion, this means that there were previously fewer than  $k$  elements in the union of all *claimed* queues and in-transit *restock* messages, so there are still fewer than  $k$ .

Thus, the total number of elements in all processes' *claimed* queues and all in-transit *restock* messages will be less than or equal to  $k$ . ◀

► **Lemma 7.** *For any prefix  $\rho$  of the sequence  $\pi$  defined in Construction 1, after locally executing  $\rho$ , every process' *size* and *clean* variables will have the same values.*

**Proof.** We first note that both *size* and *clean* are only edited in the function EXECUTELOCALLY, so we restrict our attention to that function and prove this lemma by induction on  $|\rho|$ , the length of the prefix  $\rho$ . When  $|\rho| = 0$ , all processes' variables hold their initial value of *clean* = *true* and *size* = 0.

Assume that after locally executing a prefix  $\rho'$  of length  $k$ , the claim holds. Then when any process locally executes the next operation instance  $op$  in  $\pi$ , it will follow the same logic, since EXECUTELOCALLY is deterministic and all processes have the same parameters, since they are executing the same operation instance, they will set *clean* to the same value and change *size* in the same way. The only differences in behavior that may occur at different processes are the results of the process id checks in lines 28 and 35, which do not have any effect on the values of *clean* or *size*. Thus, after executing a prefix of length  $k + 1$ , every process will have the same values for its *clean* and *size* variables. ◀

► **Lemma 8.** *At any time, for any element  $c$  in any process' *claimed* queue or in-transit *restock* message and any element  $s$  in any process' *stored* queue,  $c$  was the argument of an *Enqueue* instance which appears in the sequence  $\pi$  defined in Construction 1 before the *Enqueue* instance with  $s$  as argument.*

**Proof.** Suppose in contradiction that an element  $x$  in some process  $p_i$ 's *claimed* queue was the argument of an *Enqueue* instance  $enq = \text{Enqueue}(x)$  that appears in  $\pi$  after another instance  $enq' = \text{Enqueue}(y)$ , where  $y$  is in some process  $p_j$ 's *stored* queue (note that  $i$  and  $j$  may not be distinct). As before, there are two possible ways the algorithm may have put  $x$  in  $p_i$ 's *claimed* queue: directly by  $enq$  in line 30 or as a *restock* for a *Dequeue* instance in line 48 or 52.

Suppose first that  $x$  was added directly to  $p_i$ 's *claimed* queue by  $enq$ . Then when each of  $p_i$  and  $p_j$  locally executed  $enq$ , by line 28 and Lemma 7 we know that *clean* was true. Thus,  $p_j$  would only have put  $y$  in *stored* if either *clean* was false when  $p_j$  locally executed  $enq'$  and changed to true before  $p_j$  locally executed  $enq$ , or if *size* was at least  $k$  when  $p_j$  locally executed  $enq'$  but less than  $k$  when  $p_i$  locally executed  $enq$  (or both). *clean* could not have been false when processes locally executed  $enq'$  and true when they locally executed  $enq$  without  $y$  having been removed from *stored* and returned by a *Dequeue* in between, since *clean* is only set to true when there are no elements left in the queue, by Line 38. On the other hand, if *size* was at least  $k$  when  $p_j$  locally executed  $enq'$ , but was not when  $p_i$  locally executed  $enq$ , there must have been a *Dequeue* instance between  $enq'$  and  $enq$  in  $\pi$ , since *size* only decreases in line 37. But when each process locally executed that *Dequeue* instance, they would have set *clean* to false in line 38, so when  $p_i$  locally executed  $enq$ , it would not have stored  $x$  in *claimed*, unless *clean* was reset to true between the local execution of the *Dequeue* instance, which we have already argued could not happen. Thus,  $x$  cannot have been added to  $p_i$ 's *claimed* by  $enq$ .



The other possible way for  $x$  to be in  $p_i$ 's *claimed* queue is for it to have been put in some process  $p_k$ 's *stored* queue and passed to  $p_i$  as a restock element for a *Dequeue* instance  $deg$ . But, since all processes locally execute all instances in the same order, restocks are taken from storage in the order in which they were added (proving the claim for elements in in-transit *restock* messages), and  $y$  would be added to *stored* before  $x$ , by Lemma 4, we can conclude that  $y$  was removed from  $p_j$ 's *stored* queue by an instance  $deg'$  with a lower timestamp than that which caused  $x$  to be removed. Because the instance  $deg'$  removing  $y$  from *stored* had a lower timestamp than  $deg$ , it must have been locally executed within  $d + u + \epsilon$  time after the invocation of  $deg$ , at the latest when the timer set in line 18 (upon arrival of the message containing  $deg$ ) expires. However, no process can call EXECUTERESTOCK for  $deg$  and add  $x$  to *claimed* until the timer on line 18 expires for  $deg$  or another instance with larger timestamp. Such an instance can be invoked at most  $\epsilon$  real time before  $deg$ , and the message sent at its invocation must take at least  $d - u$  times. Thus, this instance's *restock* timer can expire no earlier than  $(d - u) + (d + 2u + \epsilon) - \epsilon = 2d + u$  real time after the invocation of  $deg$ . Thus,  $p_j$  must have locally executed  $deg'$  and removed  $y$  from *stored* at least  $(2d + u) - (d + u + \epsilon) = d - \epsilon > 0$  real time before  $p_i$  added  $x$  to *claimed*, contradicting our assumption.

Thus, for any  $x$  enqueued by an instance with smaller timestamp than that enqueueing  $y$ ,  $x$  cannot be in any process' *claimed* queue or an in-transit *restock* message while  $y$  is in another process' *stored* queue, and we have the claim. ◀

► **Lemma 9.** *The sequence  $\pi$  defined in Construction 1 is legal by the specification of a queue with Out-of-Order  $k$ -relaxed Dequeue.*

**Proof.** We prove this by induction on the length of a prefix  $\rho$  of  $\pi$ . The empty sequence is legal, proving the base case. Suppose now that  $\rho = \sigma \cdot op$  and  $\sigma$  is a legal sequence. Denote the process invoking  $op$  as  $p_h$ .

- **Case 1:**  $op = Enqueue(arg, -)$ .  $\rho$  is legal by the type specification.
- **Case 2:**  $op = Dequeue(-, retVal), retVal \in V$ . We consider the cases of fast and slow *Dequeue* instances separately:
  - Suppose  $op$  is a *fastDeq* instance. Then its return value is chosen from  $p_h$ 's *claimed* queue. By Lemmas 6, there are no more than  $k$  elements in all processes' *claimed* queues and in-transit *restock* messages. By Lemma 8, all *claimed* elements and those carried by in-transit *restock* messages are the arguments of the earliest unmatched *Enqueue* instances in the prefix of  $\pi$  before  $op$ . Thus, at  $op$ 's invocation  $retVal$  is the argument of one of the first  $k$  unmatched *Enqueue* instances. Every *Dequeue* instance's return value in a *claimed* queue is removed as soon as it is chosen (lines 7, 50) and  $p_h$  removes  $retVal$  from its *claimed* queue during  $op$ 's invocation. Also, no element is ever in more than one *claimed* queue, by the checks on lines 48 and 52 and the fact that elements are removed from *stored* when sent to *claimed* (line 35). Thus, no other *Dequeue* will return  $retVal$ . Further, once an *Enqueue* instance is among the first  $k$  unmatched, it will continue to be until it is matched, so  $\rho$  is legal.
  - Suppose  $op$  is a *slowDeq* instance. If  $op$  chooses its return value in line 50, then Lemmas 6 and 8 show that  $retVal$  is the argument of one of the first  $k$  unmatched *Enqueue* instances in  $\sigma$ , so  $\rho$  is legal. If  $op$  chooses its return value in line 51, then that value was carried by a *restock* message. As discussed before, that means it was the argument of one of the first  $k$  unmatched *Enqueue* instances, and thus  $\rho$  is legal.



- **Case 3:**  $op = Dequeue(-, \perp)$ . For a *Dequeue* instance to return  $\perp$ , then it must have chosen its return value in line 51, and have received a  $\perp$  as a restock element. For the *restock* message to have been carrying  $\perp$ , then it must have come from a process with an empty *stored* queue, in line 35. This means that when processes locally executed this *Dequeue* instance, there were no elements in any process' *stored* queue, since elements are removed from *stored* queues in the order in which they were added. This means there were fewer than  $k$  unmatched *Enqueues*, since processes locally execute all instances in the order given by  $\pi$ , by Lemma 4 and there are fewer than  $k$  elements in *claimed* elements and *restock* messages—which are the only other places values can be—by Lemma 6. Thus, by the specification of a queue with Out-of-Order  $k$ -relaxed *Dequeue*,  $\rho$  is legal. ◀

► **Theorem 10.** *Algorithm 1 is a correct, linearizable implementation of a queue with Out-of-Order  $k$ -relaxed *Dequeue*.*

**Proof.** By Lemma 5,  $\pi$  is an ordering of all operation instances which respects the real time order of non-overlapping instances and by Lemma 9,  $\pi$  is legal. Thus, for any run  $R$  of Algorithm 1, there is a linearization of  $R$ , and we have the claim. ◀

## 4.3 Complexity

### 4.3.1 Time

When discussing time complexity, we are interested in the time the algorithm takes to respond to operation invocations, in terms of the system's message timing parameters. With relaxation, we can have many *Dequeue* instances return much faster than the worst-case, leading to a low average cost for *Dequeue*. Thus, we also measure the amortized, or worst-case of the average, time required for *Dequeue*. We do not consider the amortized cost of *Enqueue* since every instance takes the (low) worst-case time.

One additional wrinkle in measuring the time complexity is that the mechanism for accelerating fast *Dequeue* instances depends on having a significant number of elements in the queue at all times. This would be the most common use case, and the number of fast *Dequeues*, and thus average performance, scales cleanly with the size of the queue, but makes general analysis difficult. We thus present bounds for the heavily-loaded case, where there are consistently at least  $k$  elements in the queue. In more lightly loaded scenarios, where there are fewer than  $k$  elements to distribute, the algorithm behaves as if  $k$  was decreased—the structure is less relaxed. Practically, the enqueue elements are distributed evenly among all processes, and they can dequeue those quickly before trying to claim more. Since *Enqueue* instances only claim elements while the structure is clean, this is the same as if  $k$  was the size of the queue at the first *Dequeue* instance until the queue is clean again. This means that relaxation scales cleanly with the queue's size when the first *Dequeue* instance occurs, up to  $k$ .

Finally, we observe that our restocking mechanism gives the possibility of much better average performance than the worst case represented in the amortized cost. A *Dequeue* instance is fast if its invoking process has claimed elements available, so slow *Dequeue* instances occur when many fast *Dequeues* in a row deplete the process' stock of claimed elements. Because we restock in the background, if *Dequeue* invocations are sufficiently infrequent, then a process will never run out of claimed elements, and all *Dequeue* instances will be fast. Exactly how infrequent *Dequeue* invocations must be for this to occur depends on the system parameters, but the average time for each would be the same as the amortized cost of the mix of fast and slow *Dequeues* which results from invoking them continuously. Thus, for the common case when many *Dequeues* are not invoked immediately one after another, a process will experience only *Dequeue* instances with low response times.

► **Theorem 11.** *The worst-case operation times for Algorithm 1 are  $\epsilon$  for Enqueue and  $2d + u + \epsilon$  for Dequeue.*

**Proof.** *Enqueue* instances and *Dequeue* instances at a process which currently has claimed elements are fast operations, and respond  $\epsilon$  time after invocation, by the timers set in lines 4 and 10 and handled in lines 12-14, while their effects are propagated through the system in the background. *Dequeue* instances at processes which do not have any claimed elements are slow, and cannot respond until the process can coordinate with other processes to claim an element. Such a *slowDeq* instance must wait for its announcement message to arrive at the process which holds the next stored element (in FIFO order), up to  $u + \epsilon$  time (line 20) for that process to ensure that it is executing instances in timestamp order, and a second message delay for that process to send the newly-claimed element back (line 35). The *slowDeq* instance can then return either the newly claimed element or one its invoking process claimed in the background execution of another operation instance while it was waiting. This delay is managed by the timer set in line 18, which starts after the  $d - u$  delay for a message from the invoking process to reach itself. Thus, the total worst-case time complexity for *Dequeue* is  $2d + u + \epsilon$ . ◀

This worst-case cost is higher than the  $d + \epsilon$  achievable in unrelaxed queues [11], but slow *Dequeue* instances are relatively infrequent, so we still obtain a low amortized cost. More concerning is the fact that we have more than doubled the worst-case cost from the algorithm for queues with out-of-order  $k$ -relaxed *Dequeue* in [9]. This is the tradeoff for reducing space complexity, and is unavoidable since at least some *Dequeue* instance must retrieve its return value from another process, taking minimum of  $2d$  time.

For amortized response time, consider a *heavily-loaded* run, defined as a run which starts with at least  $k$  *Enqueue* instances, after which there are never fewer than  $k$  unmatched *Enqueue* instances. Because there have been no *Dequeue* instances so far, when processes locally execute each of the initial  $k$  *Enqueue* instances, the check in line 29 will pass, and one process will the enqueued element. Thus, there will be  $k$  claimed elements, evenly distributed among the processes, when the first *Dequeue* is invoked. Since we assume  $k \geq n$ , this means that the first *Dequeue* will be fast, as will subsequent *Dequeue* instances until some process fast *Dequeues*  $\ell$  elements and empties its *claimed* queue. If that process invokes another *Dequeue* before restocking, it will be a slow *Dequeue* instance, and take the worst-case time of  $2d + u + \epsilon$ . By the time that slow *Dequeue* instance returns, all restocking for previous fast *Dequeue* instances will be complete, and the process will again have  $\ell$  claimed elements. In a heavily-loaded run, this pattern is the worst-case for every process, since there will always be elements to restock processes' *claimed* queues. Thus, at most one in every  $\ell$  *Dequeue* instances will be slow (recall  $\ell = \lfloor k/n \rfloor$ ) and the amortized time complexity of *Dequeue* is at most  $\frac{2d+u+\epsilon+(\ell-1)\epsilon}{\ell} = \frac{2d+u}{\ell} + \epsilon$ .

► **Theorem 12.** *The amortized time complexity of Dequeue in Algorithm 1 in a heavily-loaded run is  $\frac{2d+u}{\ell} + \epsilon$ .*

[9] gives a lower bound of  $d(1 - 1/n)$  for the amortized complexity of unrelaxed *Dequeue*, so for  $\ell > 3$ , our algorithm is faster than an unrelaxed queue, while using a factor of  $n$  less space. We also note that existing algorithms for more complex relaxations already have a similar  $2d$  term in their amortized cost and expect that we will be able to extend the benefits of this paper's work to such relaxations without significantly increasing the time cost.

### 4.3.2 Space

For space complexity, because we use a round-robin method to evenly distribute stored elements across the various processes, each process only has to hold  $(1/n)^{th}$  of the elements stored at any time. To present this in a way useful for building a system or determining whether one has the capability necessary to participate in this algorithm, we can phrase this in terms of the maximum size of the queue at any point in a run,  $T$ .

► **Theorem 13.** *Our algorithm requires memory for at most  $T/n + k/n + O(1)$  elements at any process at a time, excluding message buffers.*

**Proof.** No more than  $k$  elements are in the union of all processes' *claimed* queues at a time, by Lemma 6. These elements are evenly distributed by the round-robin procedure for claiming elements at enqueue time, or restocking elements removed from that balanced state. Thus, each process stores up to  $\lfloor k/n \rfloor + 1$  claimed elements. The restocking procedure for *Dequeue* instances attempts to keep each process at this level, but in the event of many *Dequeue* invocations in close succession, the claimed elements could be depleted, leaving all elements in storage. Because elements are stored and removed from storage in a FIFO, round-robin fashion, the number of elements stored at each process will differ by at most 1. This follows from the fact that if there have been  $E$  *Enqueue* instances and  $D$  *Dequeue* instances, each process will have stored at most  $\lfloor E/n \rfloor + 1$  elements and removed at least  $\lfloor D/n \rfloor$  of those, leaving it with less than or equal to  $\frac{E-D}{n} + 1$  elements in *stored*. Since  $T = E - D$ , we have at most  $\lfloor T/n \rfloor + 1$  stored elements at each process for a total of at most  $T/n + k/n + 2$  elements in memory.

This analysis is slightly oversimplified since processes, while they locally execute all operation instances in the same order, may not do so at the same time. It is thus possible that some process has not yet locally executed a *Dequeue* instance and removed an element from storage at the real time when an *Enqueue* instance with later timestamp adds its argument to another process' storage. This leads to at most a constant number of additional stored elements per process, however, as any one process cannot store elements from *Enqueues* with later timestamps until it has completed locally executing all previous operations. Thus, there can be at most  $n - 1$  of these extra elements, each stored at a different process, before the delayed local execution of a *Dequeue* removes an element from storage. This is less than 1 extra element stored at each process, preserving the bound. ◀

This reduction by a factor of  $n$  makes shared queues far more practical for computing devices with limited memory, as well as much more attractive to users of larger systems since a process needs only store elements which it expects to return and a share of elements which it may yet use. In a particular system, it may be possible to use a more finely-tuned storage strategy to demand more storage from processes with more resources, but we here treat the general case by balancing the load evenly.

Extending this algorithm to keep more than one copy of each data element, which would be an important component of fault-tolerance, would be simple, as we would just alter the logic in line 28 so that more than one process stored each element and in line 35 to make sure that all copies were removed from storage. The logic could be tuned to provide as much or little replication as desired, but we leave the details to ongoing work that treats all the concerns of failure tolerance.

## 5 Conclusion and Ongoing Work

We have shown that relaxing a data type can not only enable faster implementations of that type, but also reduce the type's space complexity. This is a large step towards making these data types practical in real-world systems. Now that we have shown the possibility of reducing the space complexity to a single stored copy of each data element, we know we have the flexibility to replicate them as many times as we may need for fault tolerance or resilience to poor message delays.

One other approach which could provide this level of space efficiency is a system with a centralized storage server, which coordinates requests and allocates elements to all processes. Traditionally, the two primary drawbacks to such an approach are increased communication time, since a round trip is required, and lack of fault tolerance, as a loss of the central server is a loss of all data. The first complaint does not apply, since we also require a round-trip delay for slow *Dequeues*. Since we assume no failures, we do not directly need fault tolerance. This work is not our end goal, though, as we are working towards fault-tolerant implementations, so we want to avoid structures which will make that extension more difficult.

Our end goal is to obviate all of the model assumptions that are unrealistic, eventually yielding a practical implementation of our shared queue implementation. This will involve not only space efficiency, but fault-tolerance, independence from exact knowledge of message bounds, and accounting for local computation time, at a minimum. We have addressed one of these dimensions here, and are working on the others independently. We also hope to generalize and extend results to more relaxations of more data types, creating a large collection of efficient shared data structures to aid developers of distributed systems.

Specifically, the *lateness* and *restricted Out-of-Order* relaxations of *Dequeue* are natural targets. Lateness corrects the issue with Out-of-Order relaxations that a single element at the head of the queue may be starved indefinitely by requiring that one in every  $k$  *Dequeue* instances returns the oldest element in the queue, but places no restrictions on the return values of other *Dequeue* instances. Restricted Out-of-Order combines the lateness and Out-of-Order relaxations, requiring that every *Dequeue* returns one of the  $k$  elements which were oldest when the queue's head was last returned, which means every removed element is near the head and at least one in every  $k$  *Dequeue* instances returns the head. A space-efficient implementation of a queue with a lateness  $k$ -relaxed *Dequeue* does not need to claim elements, only to coordinate removal of the head. An implementation of a queue with restricted Out-of-Order  $k$ -relaxed *Dequeue* would combine that coordination system with the claiming system of this paper's algorithm. Both implementations should be possible with the same time complexity as we achieve for the Out-of-Order relaxation.

---

### References

- 1 Yehuda Afek, Guy Korland, and Eitan Yanovsky. Quasi-linearizability: Relaxed consistency for improved concurrency. In Chenyang Lu, Toshimitsu Masuzawa, and Mohamed Mosbah, editors, *Principles of Distributed Systems - 14th International Conference, OPODIS 2010, Tozeur, Tunisia, December 14-17, 2010. Proceedings*, volume 6490 of *Lecture Notes in Computer Science*, pages 395–410. Springer, 2010. doi:10.1007/978-3-642-17653-1\_29.
- 2 Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, 1994. doi:10.1145/176575.176576.
- 3 Iwan Briquemont, Manuel Bravo, Zhongmiao Li, and Peter Van Roy. Conflict-free partially replicated data types. In *7th IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2015, Vancouver, BC, Canada, November 30 - December 3, 2015*, pages 282–289. IEEE Computer Society, 2015. doi:10.1109/CloudCom.2015.81.

- 4 Jean-Michel Héлары and Alessia Milani. About the efficiency of partial replication to implement distributed shared memory. In *2006 International Conference on Parallel Processing (ICPP 2006)*, 14-18 August 2006, Columbus, Ohio, USA, pages 263–270. IEEE Computer Society, 2006. doi:10.1109/ICPP.2006.15.
- 5 Thomas A. Henzinger, Christoph M. Kirsch, Hannes Payer, Ali Sezgin, and Ana Sokolova. Quantitative relaxation of concurrent data structures. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 317–328. ACM, 2013. doi:10.1145/2429069.2429109.
- 6 Martha J. Kosa. Time bounds for strong and hybrid consistency for arbitrary abstract data types. *Chicago Journal of Theoretical Computer Science*, 1999, 1999. URL: <http://cjtcs.cs.uchicago.edu/articles/1999/9/contents.html>.
- 7 Jennifer Lundelius and Nancy A. Lynch. An upper and lower bound for clock synchronization. *Information and Control*, 62(2/3):190–204, 1984. doi:10.1016/S0019-9958(84)80033-9.
- 8 Marios Mavronicolas and Dan Roth. Linearizable read/write objects. *Theoretical Computer Science*, 220(1):267–319, 1999. doi:10.1016/S0304-3975(98)90244-4.
- 9 Edward Talmage and Jennifer L. Welch. Improving average performance by relaxing distributed data structures. In Fabian Kuhn, editor, *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings*, volume 8784 of *Lecture Notes in Computer Science*, pages 421–438. Springer, 2014. doi:10.1007/978-3-662-45174-8\_29.
- 10 Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009. doi:10.1145/1435417.1435432.
- 11 Jiaqi Wang, Edward Talmage, Hyunyoung Lee, and Jennifer L. Welch. Improved time bounds for linearizable implementations of abstract data types. *Information and Computation*, 263:1–30, 2018. doi:10.1016/j.ic.2018.08.004.
- 12 Zhuolun Xiang and Nitin H. Vaidya. Partially replicated causally consistent shared memory: Lower bounds and an algorithm. In Peter Robinson and Faith Ellen, editors, *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 425–434. ACM, 2019. doi:10.1145/3293611.3331600.

# Recoverable, Abortable, and Adaptive Mutual Exclusion with Sublogarithmic RMR Complexity

Daniel Katzan

Tel Aviv University, Israel

Adam Morrison 

Tel Aviv University, Israel

---

## Abstract

We present the first recoverable mutual exclusion (RME) algorithm that is simultaneously abortable, adaptive to point contention, and with sublogarithmic RMR complexity. Our algorithm has  $O(\min(K, \log_W N))$  RMR passage complexity and  $O(F + \min(K, \log_W N))$  RMR super-passage complexity, where  $K$  is the number of concurrent processes (point contention),  $W$  is the size (in bits) of registers, and  $F$  is the number of crashes in a super-passage. Under the standard assumption that  $W = \Theta(\log N)$ , these bounds translate to worst-case  $O(\frac{\log N}{\log \log N})$  passage complexity and  $O(F + \frac{\log N}{\log \log N})$  super-passage complexity. Our key building blocks are:

- A  $D$ -process abortable RME algorithm, for  $D \leq W$ , with  $O(1)$  passage complexity and  $O(1 + F)$  super-passage complexity. We obtain this algorithm by using the Fetch-And-Add (FAA) primitive, unlike prior work on RME that uses Fetch-And-Store (FAS/SWAP).
- A generic transformation that transforms any abortable RME algorithm with passage complexity of  $B < W$ , into an abortable RME lock with passage complexity of  $O(\min(K, B))$ .

**2012 ACM Subject Classification** Theory of computation → Shared memory algorithms

**Keywords and phrases** Mutual exclusion, recovery, non-volatile memory

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2020.15

**Related Version** The full version of the paper is available at <http://arxiv.org/abs/2011.07622>.

**Funding** Israel Science Foundation (grant No. 2005/17) and Blavatnik ICRC at TAU.

## 1 Introduction

Mutual exclusion (ME) [10] is a central problem in distributed computing. A mutual exclusion algorithm, or *lock*, ensures that some *critical section* of code is accessed by at most one process at all times. To enter the critical section (CS), a process first executes an *entry section* to *acquire* the lock. After leaving the critical section, the process executes an *exit section* to *release* the lock. The standard complexity measure for ME is *remote memory references* (RMR) complexity [3, 6]. RMR complexity models the property that memory access cost on a shared-memory machine is not uniform. Some accesses are *local* and cheap, while the rest are *remote* and expensive (e.g., processor cache hits and misses, respectively). The RMR complexity measure thus charges a process only for remote accesses. There are various RMR definitions, modeling cache-coherent (CC) and distributed shared-memory (DSM) systems. The complexity of a ME algorithm is usually defined as its *passage complexity*, i.e., the number of RMRs incurred by a process as it goes through an entry and corresponding exit of the critical section.

For decades, the vast majority of mutual exclusion algorithms were designed under the assumption that processes are *reliable*: they do not crash during the mutual exclusion algorithm or critical section. This assumption models the fact that when a machine or program crashes, its memory state is wiped out. However, the recent introduction of non-volatile main memory (NVRAM) technology can render this assumption invalid. With NVRAM,



© Daniel Katzan and Adam Morrison;

licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 15; pp. 15:1–15:16

Leibniz International Proceedings in Informatics



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

memory state can remain persistent over a program or machine crash. This change creates the *recoverable mutual exclusion* (RME) problem [13], of designing an ME algorithm that can tolerate processes crashing and returning to execute the algorithm. In RME, a *passage* of a process  $p$  is defined as the execution fragment from when  $p$  enters the lock algorithm and until either  $p$  completes the exit section or crashes. If  $p$  crashes mid-passage and recovers, it re-enters the lock algorithm, which starts a new passage. Such a sequence of  $p$ 's passages that ends with a crash-free passage (in which  $p$  acquires and releases the lock) is called a *super-passage* of  $p$ .

RME constitutes an exciting clean slate for ME research. Over the years, locks with many desired properties (e.g., fairness) were designed and associated complexity trade-offs were explored [27]. These questions are now re-opened for RME, which has spurred a flurry of research [7, 9, 11–14, 18–21]. In this paper, we study such questions. In a nutshell, we introduce an RME algorithm that is *abortable*, *adaptive*, and has *sublogarithmic* RMR complexity. Our lock is the first RME algorithm adaptive to the number of concurrent processes (or *point contention*) and the first *abortable* RME algorithm with sublogarithmic RMR complexity. It is also the first deterministic, worst-case sublogarithmic abortable lock in the DSM model (irrespective of recoverability). Our algorithm also features other desirable properties not present in prior work, as detailed shortly.

**Abortable ME & RME.** An *abortable* lock [17, 25, 26] allows a process waiting to acquire the lock to give up and exit the lock algorithm in a finite number of its own steps. Jayanti and Joshi [21] argue that abortability is even more important in the RME setting. The reason is that a crashed process might delay waiting processes for longer periods of time, which increases the motivation for allowing processes to abort their lock acquisition attempt and proceed to perform other useful work.

Mutual exclusion, and therefore abortable ME (AME), incurs a worst-case RMR cost of  $\Omega(\log N)$  in an  $N$ -process system with standard read, write, and comparison primitives such as Compare-And-Swap (CAS) or LL/SC [6]. This logarithmic bound is achieved for both ME [28] and AME [16], and was recently achieved for a recoverable, abortable lock by Jayanti and Joshi [21]. However, while there exists an AME algorithm with sublogarithmic worst-case RMR complexity (in the CC model) [2], no such abortable algorithm is known for RME. Moreover, Jayanti and Joshi's  $O(\log N)$  abortable RME algorithm is suboptimal in a few ways. First, its worst-case RMR complexity is logarithmic only on a *relaxed* CC model, in which a failed CAS on a variable does not cause another process with a cached copy of the variable to incur an RMR on its next access to it, which is not the case on real CC machines. Their algorithm has linear RMR complexity in the realistic, standard CC model. Second, their algorithm is starvation-free only if the number of aborts is finite.

**Adaptive ME.** A lock is *adaptive* with respect to *point contention* if its RMR complexity depends on  $K$ , the number of processes concurrently trying to access the lock, and not only on  $N$ , the number of processes in the system. Adaptive locks are desirable because they are often faster when  $K \ll N$ . There exist locks with worst-case RMR cost of  $O(\min(\log N, K))$  for both ME [15] and AME [16], but no adaptive RME algorithm is known (independent of abortability).

## 1.1 Overview of Our Results

In the following, we denote the number of crashes in a super-passage by  $F$  and the size (in bits) of the system's registers by  $W$ . We obtain three key results, which, when combined, yield the first RME algorithm that is simultaneously abortable and adaptive, with worst-case  $O(\log_W N)$  passage complexity and  $O(F + \log_W N)$  super-passage complexity, in both CC



and DSM models. Assuming (as is standard) that  $W = \Theta(\log N)$ , this translates to worst-case  $O(\frac{\log N}{\log \log N})$  passage complexity and  $O(F + \frac{\log N}{\log \log N})$  super-passage complexity. In contrast to Jayanti and Joshi’s abortable RME algorithm [21], our lock achieves sublogarithmic RMR complexity in the *standard CC* model and is unconditionally starvation-free. Our algorithm’s space complexity is a static (pre-allocated)  $O(NW \log_W N)$  memory words (which translates to  $O(\frac{N \log^2 N}{\log \log N})$  if  $W = \Theta(\log N)$ ). Jayanti and Joshi’s algorithm also uses static memory, but it relies on unbounded counters. The other sublogarithmic RME algorithms [9, 11, 18] use dynamic memory allocation, and may consume unbounded space.

**Result #1:  $W$ -process abortable RME with  $O(1)$  passage and  $O(1 + F)$  super-passage complexity (§ 3).** Our key building block is a  $D$ -process algorithm, for  $D \leq W$ . It has constant RMR cost for a passage, regardless of if the process arrives after a crash. The novelty of our algorithm is that it uses the Fetch-And-Add (FAA) primitive to beat the  $\Omega(\log D)$  passage complexity lower-bound. In contrast, the building blocks in prior RME work with worst-case sublogarithmic RMR complexity use the Fetch-And-Store (FAS, or SWAP) primitive and assume no bound on  $D$ , even though they are ultimately used by only a bounded number of processes in the final algorithm. By departing from FAS and exploiting the process usage bound, we overcome difficulties that made the prior algorithms’ building blocks [11, 18] have only  $O(D)$  RMR passage complexity.

These prior algorithms use a FAS-based queue-based lock as a building block. They start with an  $O(1)$  RMR queue-based ME algorithm [8, 23], in which a process trying to acquire the lock uses FAS to append a node to the queue tail, and then spins on that node waiting for its turn to enter the critical section. Unfortunately, if the process crashes after the FAS, before writing its result to memory, then when it recovers and returns to the algorithm, it does not know whether it has added itself to the queue and/or who is its predecessor (previously obtained from the FAS response). To overcome this problem, a recovering process reconstructs the queue state into some valid state, *which incurs a linear number of RMRs*. The recovery procedure is blocking (not wait-free), and multiple processes cannot recover concurrently. Overall, these prior building blocks have  $O(D)$  passage complexity and  $O(1 + FD)$  super-passage complexity. In contrast, our  $D$ -process abortable RME algorithm has  $O(1)$  passage complexity and  $O(1 + F)$  super-passage complexity, has wait-free recovery, and allows multiple processes to recover concurrently. While other  $O(1)$  RME algorithms exist, they either assume a weaker crash model [12], rely on non-standard primitives that are not available on real machines [11, 19], or obtain only amortized, not worst-case,  $O(1)$  RMR complexity [7].

**Result #2: Tournament tree with wait-free exit (§ 4).** In both ours and prior work [11, 18], the main lock is obtained by constructing a *tournament tree* from the  $D$ -process locks. The tree has  $N$  leaves, one for each process. Each internal node is a  $D$ -process lock, so the tree has height  $O(\log_D N)$ . To acquire the main lock, a process competes to acquire each lock on the path from its leaf to the root, until it wins at the root and enters the critical section. Our algorithm differs from prior tournament trees in a couple of simple ways, but which have important impact.

First: In our tree, a process that recovers from a crash returns directly to the node in which it crashed. This allows us to leverage our node lock’s  $O(1 + F)$  super-passage complexity to obtain  $O(H + F)$  super-passage complexity for the tree, where  $H$  is the tree’s height. By taking  $D = W = \Theta(\log N)$ , our overall lock has  $O(F + \frac{\log N}{\log \log N})$  super-passage complexity and  $O(\frac{\log N}{\log \log N})$  passage complexity. In contrast, prior trees perform recovery by having a process restart its ascent from the leaf. In fact, in these algorithms, there is no asymptotic benefit from returning directly to the node where the crash occurred. The reason is that

■ **Table 1** Comparison of RME algorithms. (SP: super-passage, WF: wait-free,  $F^*$ : total number of crashes in the system, FASAS: Fetch-And-Swap-And-Swap.) All algorithms satisfy starvation-freedom, wait-free critical-section re-entry, and wait-free exit (defined in § 2).

Algorithm	Passage Complexity	Super-Passage Complexity	Primitives Used	Space Complexity	Additional Properties
Golab & Ramaraju [14, Section 4.2] with MCS [23] as base lock	$O(1)$ (no concurrent crashes) ----- $O(\log N)$ (concurrent crashes) ----- $O(N)$ (if crashes)	$O(1)$ (no concurrent crashes) ----- $O(\log N)$ (concurrent crashes) ----- $O(FN)$ (if crashes)	CAS, FAS	$O(N \log N)$	
Jayanti & Joshi [20]	$O(\log N)$	$O(\log N + F)$	CAS	$O(N \log N)$	FCFS, SP WF Exit
Jayanti, Jayanti, & Joshi [18]	$O(\frac{\log N}{\log \log N})$	$O((1 + F)(\frac{\log N}{\log \log N}))$	FAS	Unbounded	
Jayanti, Jayanti, & Joshi [19]	$O(1)$	$O(1)$ in the DSM model $O(F)$ in the CC model	FASAS	$O(N)$	SP WF Exit
Chan & Woelfel [7]	$O(1)$ amortized	same as passage complexity	CAS, FAA	Unbounded	SP WF Exit
Dhoked & Mittal [9]	$O(\min(\sqrt{F^*}, \frac{\log N}{\log \log N}))$	same as passage complexity	CAS, FAS	Unbounded	Crash-adaptive
Jayanti & Joshi [21]	$O(\log N)$	$O(\log N + F)$	CAS	$O(N \log N)$	Abortable, SP WF Exit, FCFS
This work	$O(\min(K, \frac{\log N}{\log \log N}))$	$O(\min(K, \frac{\log N}{\log \log N}) + F)$	FAA, CAS	$O(\frac{N \log^2 N}{\log \log N})$	Abortable, adaptive, SP WF Exit

node lock recovery in these trees has  $O(D)$  complexity, so to obtain overall sublogarithmic complexity, they take  $D = \frac{\log N}{\log \log N}$ , which means that node crash recovery costs the same as climbing to the node. Consequently, their overall super-passage complexity is multiplicative in  $F$ ,  $O((1 + F)\frac{\log N}{\log \log N})$ , instead of additive as in our tree.

Second: Our tree’s exit section is wait-free (assuming finitely many crashes). In contrast, in the prior trees, a process that crashes during its exit section might subsequently block. The reason is a subtle issue related to composition of RME locks. The model in these works [11, 18] is that a process  $p$  that crashes in its exit section must complete a crash-free passage upon recovery (i.e., re-enter the critical section and exit it again). Thus,  $p$  must re-ascend to the root after recovering. Each node lock satisfies a *bounded CS re-entry* property, which allows  $p$  to re-enter the node’s CS (i.e., ascend) without blocking – provided that  $p$  crashed inside the node’s CS. However, this property does not apply if  $p$  released the node lock (i.e., descended) before crashing. For such a node,  $p$  simply attempts to re-acquire the node lock. Consequently,  $p$  might block during its recovery, even though logically *it is only trying to release the overall lock*. We address this problem by carefully modeling the interface of an RME algorithm in a way that facilitates composition, which enables a recovering process to avoid re-acquiring node locks it had already released. Our overall algorithm thereby satisfies a new *super-passage wait-free exit* property.

**Result #3: Generic RME adaptivity transformation (§ 5).** We present a generic transformation that transforms any abortable RME algorithm with passage complexity of  $B < W$  into an abortable RME lock with passage complexity of  $O(\min(K, B))$ , where  $K$  is the number of processes executing the algorithm concurrently with the process going through the super-passage, i.e., the *point contention*. Applying this transformation to our tournament tree lock yields the final algorithm.

**Summary of contributions and related work.** Table 1 compares our final algorithm to prior RME work. Dhoked and Mittal [9] use a definition of “adaptivity” that requires RMR cost to depend on the total number of crashes; we refer to this property as *crash-adaptivity*. Crash-adaptivity is thus orthogonal to the traditional notion of adaptivity [5]. Chan & Woelfel’s algorithm [7] uses FAA, but it is used to assign processes with tickets, which is different from our technique (§ 3). Their algorithm has only an amortized RMR passage complexity bound and its worst-case RMR cost is unbounded.

## 2 Model and Preliminaries

**Model.** We consider a system in which  $N$  deterministic, asynchronous, and unreliable processes communicate over a shared memory. The shared memory,  $M$ , is an array of  $\Theta(W)$ -bit words. (Henceforth, we refer to the shared memory simply as “memory”; process-private variables are not part of the shared memory.) The system supports the standard read, write, CAS, and FAA operations.  $CAS(a, o, n)$  atomically changes  $M[a]$  from  $o$  to  $n$  if  $M[a] = o$  and returns true; otherwise, it returns false without changing  $M[a]$ .  $FAA(a, x)$  atomically adds  $x$  to  $M[a]$  and returns  $M[a]$ ’s original content.

A *configuration* consists of the state of the memory and of all processes, where the state of process  $p$  consists of its internal program counter and (non-shared) variables. Given a configuration  $\sigma$ , an *execution fragment* is a (possibly infinite) sequence of *steps*, each of which moves the system from one configuration to another, starting from  $\sigma$ . In a *normal step*, some process  $p$  invokes an operation on a memory word and receives the operation’s response. In a *crash step*, the state of some process  $p$  resets to its initial state (but the memory state remains unchanged). An *execution* is an execution fragment starting from the system’s initial configuration.

**Notation.** Given an execution fragment  $\alpha$ , if  $\beta$  is a subsequence of  $\alpha$ , we write  $\beta \subseteq \alpha$ . If  $e$  is a step taken in  $\alpha$ , we write  $e \in \alpha$ . If  $e$  is the  $t$ -th step in an execution  $E$ , we say that  $e$  is at time  $t$ . We use  $[t, t']$  to denote the subsequence of  $E$  whose first and last steps are at times  $t$  and  $t'$  in  $E$ , respectively.

**RMR complexity.** The RMR complexity measure breaks the memory accesses by a process  $p$  into *local* and *remote* references, and charges  $p$  only for remote references. We consider two types of RMR models. In the DSM model, each memory word is local to one process and remote to all others, and process  $p$  performs an RMR if it accesses a memory word remote to it. In the CC model, the processes are thought of as having coherent caches, with RMRs occurring when a process accesses an uncached memory word. Formally: (1) every write, CAS, or FAA is an RMR, and (2) a read by  $p$  of word  $x$  is an RMR if it is the first time  $p$  accesses  $x$  or if after  $p$ ’s prior access to  $x$ , another process performed a write, CAS, or FAA on  $x$ .

**Recoverable mutual exclusion (RME).** Our RME model draws from the models of Golab and Ramaraju [14] and Jayanti and Joshi [20]. In the spirit of [14], we model the RME algorithm as an object exporting methods invoked by a client process. In the spirit of [20], we require recovery to re-execute the section in which the crash occurred, rather than restart the entire passage. An *RME algorithm* (or *lock*) provides the methods *Recover*, *Try*, and *Exit*. (In the code, we show the methods taking an argument specifying the calling process’ id.) If process  $p$  invokes *Try* and it returns TRUE, then  $p$  has *acquired* the lock and *enters* the critical section (CS). Subsequently,  $p$  *exits* the CS by invoking *Exit*. If *Exit* completes, we say that  $p$  has *released* the lock. The *Recover* method guides  $p$ ’s execution after a crash, which resets  $p$  to its initial state. We assume  $p$ ’s initial state is to invoke *Recover*, which returns  $r \in \{TRY, CS, EXIT\}$ . If  $r = TRY$ ,  $p$  invokes *Try*. If  $r = CS$ ,  $p$  enters the CS. If  $r = EXIT$ ,  $p$  invokes *Exit*.

A *super-passage* of  $p$  begins with  $p$  completing *Recover* and invoking *Try*, either for the first time, or for the first time after  $p$ ’s prior super-passage ended. The super-passage ends when  $p$  completes *Exit*. A *passage* of  $p$  begins with  $p$  starting a super-passage, or when  $p$  invokes *Recover* following a crash step. The passage ends at the earliest of  $p$  completing *Exit* or crashing. We refer to an  $L$ -passage (or  $L$ -super-passage) to denote the lock  $L$  that a

passage (or super-passage) applies to; similarly, we refer to a step taken in lock  $L$ 's code as an  $L$ -step. We omit  $L$  when the context is clear. These definitions facilitate composition of RME locks. For instance, suppose that process  $p$  is releasing locks in a tournament tree and crashes after releasing some node lock  $L$ . When  $p$  recovers, it can invoke  $L.Recover$ , which will return  $TRY$ , and thereby learn that it has released  $L$  and can descend from it – without the  $Recover$  invocation counting as starting a new  $L$ -super-passage.

*Well-formed* executions formalize the above described process behavior:

► **Definition 1.** *An execution is well-formed if the following hold for every lock  $L$  and process  $p$ :*

1. Recover invocation:  $p$ 's first  $L$ -step after a crash step is to invoke  $L.Recover$ .
2. Try invocation:  $p$  invokes  $L.Try$  only if  $p$  is starting a new  $L$ -super-passage, or if  $p$ 's prior crash step was during  $L.Try$ .
3. CS invocation:  $p$  enters the CS of  $L$  only if  $p$  receives  $TRUE$  from  $L.Try$  in its current  $L$ -passage, or if  $p$ 's prior crash step was during the CS.
4. Exit invocation:  $p$  invokes  $L.Exit$  only if  $p$  is in the CS of  $L$ , or if  $p$ 's prior crash step was during  $L.Exit$ .

Henceforth, we consider only well-formed execution. We also consider only *well-behaved* RME algorithms, in which  $Recover$  correctly identifies where a process crashes:

► **Definition 2.** *An RME algorithm is well-behaved if the following hold, for every process  $p$  and every well-formed execution:*

1.  $p$ 's first complete invocation of  $Recover$ , and  $p$ 's first complete invocation of  $Recover$  following a complete passage of  $Exit$ , returns  $TRY$ .
2.  $p$ 's first complete invocation of  $Recover$  following a crash during  $Try$  return  $TRY$ .
3.  $p$ 's first complete invocation of  $Recover$  following a crash during the CS returns  $CS$ .
4.  $p$ 's first complete invocation of  $Recover$  following a crash during  $Exit$  returns  $EXIT$ .
5. A complete invocation of  $Recover$  by  $p$  during the CS returns  $CS$ .

Note: We consider  $p$  to be in the *Try* or *Exit* section from the time it executes the first memory operation of that section and until it either crashes or executes the last memory operation of that section. Thus,  $p$  is considered to be in the CS after it executes its final *Try* memory operation.

**Fairness.** We make a standard fairness assumption on executions: once  $p$  starts a super-passage, it does not stop taking steps until the super-passage ends.

**Abortable RME.** At any point during its super-passage, process  $p$  can non-deterministically choose to abort its attempt, which we model by  $p$  receiving an external *abort* signal that remains visible to  $p$  throughout the super-passage (i.e., including after crashes) and resets once  $p$  finishes the super-passages. Abortable RME extends the definition of a super-passage as follows. If  $p$  is signalled to abort and its execution of *Try* returns  $FALSE$ , then  $p$  has aborted and the super-passage ends. (It is not mandatory for *Try* to return  $FALSE$ , because an abort may be signalled just as  $p$  acquires the lock.)

**$D$ -ported locks.** We model locks that may be used by at most  $D$  processes concurrently as follows. In a  *$D$ -ported lock*, each process invokes the methods with a *port* argument,  $1 \leq k \leq D$ , which acts as an identifier. We augment the definition of a well-formed execution to include the following conditions:

5. *Constant port usage*: For every process  $p$  and  $L$ -super-passage of  $p$ ,  $p$  does not change its port for  $L$  throughout the super-passage.
6. *No concurrent super-passages*: For any  $L$ -super-passages  $sp_i$  and  $sp_j$  of processes  $p_i \neq p_j$ , if  $sp_i$  and  $sp_j$  are concurrent, then  $p_i$ 's port for  $L$  in  $sp_i$  is different than  $p_j$ 's port for  $L$  in  $sp_j$ . (Two super-passages are not *concurrent* if one ends before the other begins.)

**Problem statement.** Design a well-behaved abortable RME algorithm with the following properties.

1. **Mutual exclusion**: At most one process is in the CS at any time  $t$ .
2. **Deadlock-freedom**: If a process  $p$  starts a super-passage  $sp$  at time  $t$ , and does not abort  $sp$ , and if every process that enters the CS eventually leaves it, then there is some time  $t' > t$  and some process  $q$  such the  $q$  enters the CS in time  $t'$ , or else there are infinitely many crash steps.
3. **Bounded abort**: If a process  $p$  has abort signalled while executing *Try*, and executes sufficiently many steps without crashing, then  $p$  complete its execution of *Try*.

The following properties are also desirable, and all but FCFS are satisfied by our algorithm:

4. **Starvation-freedom**: If the total number of crashes in the execution is finite and process  $p$  executes infinitely many steps and every process that enters the CS eventually leaves it, then  $p$  enters the CS in each super-passage in which it does not receive an abort signal.
5. **CS re-entry**: If process  $p$  crashes while in the CS, then no other process enters the CS from the time  $p$  crashes to the time when  $p$  next enters the CS.
6. **Wait-free CS re-entry**: If process  $p$  crashes in the CS, and executes sufficiently many steps without crashing, then  $p$  enters the CS.
7. **Wait-free exit**: If process  $p$  is executing *Exit*, and executes sufficiently many steps without crashing, then  $p$  completes its execution of *Exit*.
8. **Super-passage wait-free exit**: If process  $p$  is executing *Exit*, then  $p$  completes an execution of *Exit* after a finite number of its own steps, or else  $p$  crashes infinitely many times. (Notice that  $p$  may crash and return to re-execute *Exit*.)
9. **First-Come-First-Served (FCFS)**: If there exists a bounded section of code in the start of the entry section, referred to as the *doorway* such that, if process  $p_i$  finishes the doorway in its super-passage  $sp_i$  for the first time before some process  $p_j$  begins its doorway for the first time in its super-passage  $sp_j$ , and  $p_i$  does not abort  $sp_i$ , then  $p_j$  does not enter the CS in  $sp_j$  before  $p_i$  enters the CS in  $sp_i$ .

Super-passage wait-free exit is a novel property introduced in this work. It guarantees that a process completes *Exit* in a finite number of its own steps, as long as it only crashes finitely many times. Wait-free exit does not imply super-passage wait-free exit since it does not apply if the process crashes during *Exit*. Clearly, starvation-freedom implies deadlock-freedom, wait-free CS re-entry implies CS re-entry, and super-passage wait-free exit implies wait-free exit.

**Lock complexity.** The *passage complexity* (respectively, *super-passage complexity*) of a lock is the maximum number of RMRs that a process can incur while executing a passage (respectively, super-passage). We denote by  $F$  the maximum number of times a process crashes in an execution.

### 3 *W*-Port Abortable RME Algorithm

Here, we present our  $D$ -process abortable RME algorithm, for  $D \leq W$ , which has  $O(1)$  passage RMR complexity and  $O(1 + F)$  super-passage complexity. The algorithm is similar in structure to Jayanti and Joshi’s abortable RME algorithm [21], in that it is built around a recoverable auxiliary object that tracks the processes waiting to acquire the lock. This object’s RMR complexity determines the algorithm’s complexity. Non-abortable RME locks implement such an object with a FAS-based linked list [11, 18]. Such a list has  $O(1 + FD)$  super-passage complexity – i.e., a crash-free passage incurs  $O(1)$  RMRs – but it is hard to make abortable. Jayanti and Joshi instead use a recoverable *min-array* [15]. This object supports aborting, but its passage complexity is logarithmic, even in the absence of crashes.

Our key idea is to represent the “waiting room” object with a FAA-based  $W$ -bit mask (a single word), where a process  $p$  arriving/leaving is indicated by flipping a bit associated with  $p$ ’s port. The key ideas are that (1) if  $p$  crashes and recovers, it can learn its state in  $O(1)$  RMRs simply by reading the bit mask and (2) the algorithm carefully avoids relying on any FAA’s return value. Our design thus obtains the best of both worlds: the object can be updated with  $O(1)$  RMRs as well as supports efficient aborting (with a single bit flip). The trade-off we make in this design choice is that we only guarantee starvation-freedom, but not FCFS. Unlike a min-array, the bit mask cannot track the order of arriving processes, as bit setting operations commute. We do, however, track the order in which processes acquire the lock, and thereby guarantee starvation-freedom.

Our algorithm guarantees starvation-freedom unconditionally, even if there are infinitely many aborts. This turns out to be a subtle issue to handle correctly (§ 3.2), and the Jayanti and Joshi algorithm is prone to executions in which a process that does not abort starves as a result of other processes aborting infinitely often (we show an example in § 3.2).

Since we assume  $W$ -bit memory words, we are careful not to use unbounded, monotonically increasing counters, which the Jayanti and Joshi lock does use. Our algorithm’s RMR bounds are in both the DSM and CC models, whereas the Jayanti and Joshi lock has linear RMR complexity on the standard CC model.

#### 3.1 Algorithm Walk-Through

Figure 1 presents the pseudo code of the algorithm. We assume participating processes uses distinct ports in the range  $0, \dots, W - 1$ , so we refer to processes and ports interchangeably. For simplicity, we present the algorithm assuming dynamic memory allocation with safe reclamation [24]. In this environment, a process can *allocate* and *retire* objects, and it is guaranteed that an allocation does not return a previously-retired object if some process still has a reference to that object. We show how to satisfy this assumption (with  $O(D^2)$  static, pre-allocated memory) in the full version [22, Appendix A].

Each process  $p$  has a status word,  $STATUS[p]$ , and a pointer to a boolean spin variable,  $GO[p]$ . (In the DSM model, a process allocates its spin variables from local memory, so that it can spin on them with  $O(1)$  RMR cost.) The lock’s state consists of a  $W$ -bit word,  $ACTIVE$ , and a  $\Theta(W)$ -bit word,  $LOCK\_STATUS$ . The  $LOCK\_STATUS$  word holds a tuple  $(taken, owner, owner\_go)$ , where *taken* is a bit indicating if the lock is acquired by some process. If *taken* is set, *owner* is the id (port) of the lock’s owner and *owner\_go* points to the owner’s spin variable.

The  $STATUS$  word of each process  $p$ , initialized to  $TRY$ , indicates in which section the process is currently at. This information is used by *Recover* to steer  $p$  to the right method when it arrives. The  $STATUS$  word changes when completing *Try* and entering the CS, when



aborting during *Try*, when exiting the CS and executing *Exit*, and when *Exit* completes. Note that the *Exit* method may be called as a subroutine during the *Try* section's abort flow. In this case, its operations are considered part of the *Try* section (i.e., the subroutine call is to avoid putting a copy of *Exit*'s code in *Try*). To distinguish these subroutine calls from when a process invokes *Exit* to exit the CS, we add an *abort* argument to *Exit*, which is *FALSE* if and only if *Exit* is invoked to exit the CS (i.e., not as a subroutine).

In the normal (crash- and abort-free) flow, a passage of process *p* proceeds as follows. First, *p* allocates its spin variable, if it does not currently exist (lines 11–16). Then *p* flips its bit in the *ACTIVE* word, but only if *p*'s bit is not already set (lines 17–18). This check avoids corrupting *ACTIVE* when *p* recovers from a crash. Next, *p* executes a *Promote* procedure, which tries to pick some waiting process (possibly *p*) and make it the owner of the lock, if the lock is currently unowned (line 19). Finally, *p* begins spinning on its spin variable, waiting for an indication that it has become the lock owner (lines 20–25). Upon exiting the CS, *p* clears its bit in *ACTIVE* (again, only if the bit is currently set, to handle crash recovery) (lines 39–40). Then *p* executes *Promote* (line 41). Performing this call will have no effect, since *p* is still holding the lock, which may appear strange, but is required in order to support the abort flow, as explained shortly. Then, if *p* is indeed the lock owner (another check useful only in the abort flow), it releases the lock by clearing the *taken* bit in *LOCK\_STATUS* (lines 42–45). Note that *p* leaves the *owner* and *owner\_go* fields intact, for reasons described shortly. Finally, *p* executes *Promote* again, to hand the lock off to some waiting process (line 46). It then retires its spin variable, clears its *GO* pointer, and updates its *STATUS* to *TRY*, thereby completing *Exit* and thus its current passage and super-passage (line 47–50).

```

1  ACTIVE: int // initially 0
2  STATUS: array of W status words // initially all TRY
3  GO: array of W pointers to booleans // initially all ⊥
4  LOCK_STATUS: struct {bool, port_id, bool*}
5  // initially (0, 0, ⊥)

7  void Try(int k) {
8      if STATUS[k] = ABORT:
9          Exit(k, TRUE)
10         return FALSE
11         if GO[k] = ⊥:
12             if got abort signal:
13                 STATUS[k] := ABORT
14                 Exit(k, TRUE)
15                 return FALSE
16                 GO[k] := new Bool()
17             if k-th bit in ACTIVE is 0:
18                 FAA(ACTIVE, 2k)
19                 Promote(⊥)
20                 while *GO[k] = FALSE:
21                     if got abort signal:
22                         STATUS[k] := ABORT
23                         Exit(k, TRUE)
24                         return FALSE
25
26                 STATUS[k] := CS
27                 return TRUE
28         }
29         status Recover(int k) {
30             if STATUS[k] = EXIT:
31                 return EXIT
32             if STATUS[k] = CS:
33                 return CS
34             return TRY
35         }

36 void Exit(int k, bool abort) {
37     if abort = FALSE:
38         STATUS[k] = EXIT
39         if k-th bit in ACTIVE is 1:
40             FAA(ACTIVE, -2k)
41         Promote(k)
42         (taken, owner, owner_go) := LOCK_STATUS
43         if taken = 1 and owner = k:
44             CAS(LOCK_STATUS, (1, owner, owner_go),
45                 (0, owner, owner_go))
46         Promote(⊥)
47         if GO[k] ≠ ⊥:
48             Retire(GO[k])
49             GO[k] := ⊥
50         STATUS[k] := TRY
51     }
52     void Promote(int j) {
53         (taken, owner, owner_go) := LOCK_STATUS
54         if taken = 0:
55             active := ACTIVE
56             if active ≠ 0:
57                 j := next(owner, active)
58             if j ≠ ⊥:
59                 CAS(LOCK_STATUS, (0, owner, owner_go),
60                     (1, j, GO[j]))
61         (taken, owner, owner_go) := LOCK_STATUS
62         if taken = 1:
63             *owner_go := TRUE
64     }

```

■ **Figure 1** *W*-port abortable RME algorithm.



If  $p$  receives the abort signal while spinning in *Try*, it sets its *STATUS* to *ABORT*, executes the *Exit* method as a subroutine, and returns *FALSE* (label 12–15). If  $p$  crashes during the execution of *Exit*, *Recover* will steer it to *Try* once it recovers, at which point it will again execute the *Exit* method and return *FALSE*. In the abort flow, the call to *Exit* does not modify  $p$ 's *STATUS* (the *if* is not taken, lines 37–38).

The main goal of *Promote*( $j$ ) is to promote some waiting process to be the lock owner, if the lock is currently unowned. *Promote* tries to promote one of the waiting processes (as specified by *ACTIVE*). If there is no such process, then *Promote* tries to promote process  $j$  if  $j \neq \perp$ , and does not promote any process otherwise (lines 53–60). A secondary goal of *Promote* is that it signals the (current or newly promoted) owner by writing to its spin variable (lines 61–63). Picking a process to promote from among the waiting processes is done in a manner that guarantees starvation-freedom. To this end, *Promote* picks the next id whose bit is set in *ACTIVE*, when ids are scanned starting from the previous owner's id (which, as described above, is written in *LOCK\_STATUS*) and moving up (modulo  $W$ ). (In the code, this is specified as  $next(owner, active)$ .) Having picked a process  $q$  to promote, *Promote* tries to update *LOCK\_STATUS* to  $(1, q, GO[q])$  using a single CAS. Finally, before completing, *Promote* checks again if the lock is owned by some process  $r$  (possibly  $r \neq q$ ), and if so, signals  $r$  by writing *TRUE* to  $r$ 's spin variable.

The reason for executing *Promote* in *Exit* before releasing the lock, and not only afterwards, is to handle a scenario in which the lock owner  $q$  has released the lock and  $next(q, ACTIVE) = p$ , so any process  $r$  (possibly, but not necessarily,  $q$ ) executing *Promote* tries to hand the lock to  $p$ . If now  $p$  is signalled to abort, and did not also execute *Promote* before departing, deadlock would occur. By having  $p$  call *Promote*( $p$ ), we guarantee that either (1) some process (possibly  $p$ ) promotes  $p$ , so  $p$ 's *Exit* call releases the lock before completing the abort; or (2) some process  $r$  (possibly, but not necessarily  $p$ ), which does not observe  $p$  in *ACTIVE*, updates *LOCK\_STATUS* from  $(0, q, G)$  to  $(1, q', G')$ . In the latter case, our memory management assumption implies that *LOCK\_STATUS* will not recycle to contain  $(0, q, G)$  before every processes that has read  $(0, q, G)$  from *LOCK\_STATUS* executes its CAS. All such CASs, who are about to change  $(0, q, G)$  to  $(1, p, GO[p])$  thus fail, so the lock does not get handed to  $p$  and no deadlock occurs after it completes its abort.

### 3.2 Discussion: Guaranteeing Starvation-Freedom In the Presence of Infinitely Many Aborts

As discussed in § 3.1, a key idea in our algorithm is to invoke *Promote* even before releasing the lock, to handle the case in which the lock is about to be handed to an aborting process. While simple, this is a subtle idea, because a different (more straightforward) approach to dealing with this issue can lead to starvation. We explain the issue by describing and analyzing a starvation problem in Jayanti and Joshi's abortable RME algorithm [21]. The structure of our algorithm and of Jayanti and Joshi's algorithm is similar, if one thinks of our *ACTIVE* word and their min-array as abortable objects which (1) maintain the set of waiting processes and (2) have some notion of the “next in line” waiting process, which becomes the lock owner. (Jayanti and Joshi refer to this object as a *registry*.) We describe the problem in the Jayanti and Joshi lock by contrasting its behavior with our algorithm's.

Intuitively, starvation-freedom should follow from property (2) of the “waiting room” object, because every process executing *Promote* will eventually agree on the process  $p$  to promote, which would then become the lock owner. For this to be true, however, aborts need to be handled very carefully. Phrased in our terminology, in the Jayanti and Joshi algorithm, a process  $p$  that receives an abort signal starts executing *Exit*, where it removes

itself from the “waiting room” object. Subsequently, if  $LOCK\_STATUS = (0, o, os)$ ,  $p$  tries (using a single CAS) to update  $LOCK\_STATUS$  from  $(0, o, os)$  to  $(0, p, GO[p])$ . In other words,  $p$  tries to make it look as if it had acquired the lock and immediately released it. The motivation for this step is to fail any *Promote* that is about to make  $p$  the lock owner, which if not handled, would result in deadlock.

This approach has the unfortunate side-effect of failing concurrent *Promotes* even if they are not about to make  $p$  the lock owner. This can lead to an execution in which aborting processes prevent the lock from being acquired, as described next.

Process  $p_1$  arrives and enters the critical section. Process  $p_2, p_3, p_4$  arrive and enter the waiting room. Now  $p_1$  leaves the CS and executes *Exit*, which (in Jayanti and Joshi’s algorithm) has a single *Promote* call, after releasing the lock. Suppose the “waiting room” object indicates that  $p_2$  should be the next lock owner. Now,  $p_1$  stops in its *Promote* call, just before CASing  $LOCK\_STATUS$  from  $(0, p_1, *)$  to  $(1, p_2, *)$ . Next,  $p_3$  aborts, executes the *Exit* code and successfully changes  $LOCK\_STATUS$  to  $(0, p_3, *)$ .

As a result,  $p_1$ ’s CAS in *Promote* fails.  $p_1$  completes its *Exit* section and then returns to the Try section, executes *Promote*, and stops just before CASing  $LOCK\_STATUS$  from  $(0, p_3, *)$  to  $(1, p_2, *)$ . Now,  $p_3$  proceeds to the *Promote* call in *Exit*, stopping just before CASing  $LOCK\_STATUS$  from  $(0, p_3, *)$  to  $(1, p_2, *)$ . We have reached a state in which  $p_4$  is waiting,  $LOCK\_STATUS$  is  $(0, p_3, *)$ ,  $p_1$  is in its Try *Promote* and  $p_3$  is in its *Exit* promote, both about to CAS  $LOCK\_STATUS$  from  $(0, p_3, *)$  to  $(1, p_2, *)$ .

We continue as follows. Now  $p_4$  receives the abort signal, proceeds to execute *Exit*, and successfully changes  $LOCK\_STATUS$  from  $(0, p_3, *)$  to  $(0, p_4, *)$ . Consequently, the CAS of both  $p_1$  and  $p_3$  fails, so  $p_1$  enters the waiting room, whereas  $p_3$  departs the algorithm, returns, and stops in the Try *Promote* before CASing  $LOCK\_STATUS$  from  $(0, p_4, *)$  to  $(1, p_2, *)$ . As for  $p_4$ , it enters the *Exit Promote* and stops before CASing  $LOCK\_STATUS$  from  $(0, p_4, *)$  to  $(1, p_2, *)$ . We have reached a similar situation as in the previous paragraph, and can therefore keep repeating this scenario indefinitely. Throughout,  $p_2$  keeps taking steps in the waiting room, but will never enter the CS.

### 3.3 Proofs of RME Properties

We refer to our  $W$ -port abortable RME algorithm as Algorithm  $M$ . In the the full version [22], we prove the following theorem:

► **Theorem 3.** *If every execution of Algorithm  $M$  is well-formed, then Algorithm  $M$  satisfies mutual exclusion, bounded abort, starvation-freedom, CS re-entry, wait-free CS re-entry, wait-free exit, and super-passage wait-free exit. The passage complexity of Algorithm  $M$  in both the CC and DSM models is  $O(1)$  and the super-passage complexity is  $O(1 + F)$ . (Assuming, for the DSM model, that process memory allocations return local memory.) The space complexity of the algorithm is  $O(D^2)$ .*

Here, we omit the proof, due to space constraints, and point out of some of its high-level aspects. Whenever a process  $p$  starts a super-passage in our algorithm, it allocates a fresh spin variable. To avoid unbounded space consumption, the memory used for spin variables eventually has to be recycled, i.e., an allocation by process  $p$  can return a variable it previously used. Our proofs assume that this recycling is done safely, namely, that an allocation of a new spin variable does not return an object that is currently being referenced by some process. (We show how to satisfy this assumption using  $O(D^2)$  static pre-allocated memory words in the full version [22, Appendix A].)

The above safe memory management assumption implies two properties that we use throughout the proofs. First, that if a process  $p$  is about to CAS  $LOCK\_STATUS$  in *Promote*, and  $LOCK\_STATUS$  has changed between  $p$  last reading it and executing the CAS, then the CAS will fail. This holds because  $LOCK\_STATUS$  necessarily contains a different *owner\_go* value. Second, that if  $p$  sets the spin variable of  $q$  to *TRUE* and  $q$  has already started a new super-passage, then  $q$  will never read that *TRUE* value. This holds because  $q$  allocates a different spin variable for its new super-passage.

## 4 Tournament Tree

A *tournament tree* lock, referred to as the *main* lock, is constructed by statically arranging multiple  $D$ -port RME algorithms, referred to as *node* locks, in a  $D$ -ary tree with  $N$  leaves (we assume  $D \leq W$ ). Each leaf is uniquely associated with a process. To acquire the main lock, a process competes to acquire each lock on the path from its leaf to the root, until it wins at the root and enters the main lock's CS. To release the main lock, the process descends from the root to its leaf, releasing each node lock on the path. In this section, we present our tournament tree algorithm.

Our algorithm has two distinguishing features: (1) that its super-passage RMR complexity is additive in  $F$ , the number of crashes, and not multiplicative; and (2) that it satisfies *super-passage wait-free exit* (SP-WF-Exit), i.e., a process releasing the main lock is guaranteed to complete some execution of *Exit* after a finite number of its own steps (including crashes).

Our algorithm's super-passage RMR complexity is  $O(FR + B \log_D N)$ , where  $R$  and  $B$  are the recovery cost and passage complexity of the node lock, respectively. In comparison, prior trees have super-passage complexity of  $O(F(R + B \log_D N))$ . Obtaining our bound is simple: a process just needs to write its location in the tree to NVRAM, so that upon crash recovery, it can resume from there instead of starting to walk up or down the tree from scratch. We suspect that this simple optimization was not performed in prior tournament trees because their node lock has  $R = \log_D N = O(\frac{\log N}{\log \log N})$  and  $B = O(1)$ , so directly returning to the node at which the crash occurred does not asymptotically improve complexity. With our  $W$ -port RME algorithm, however,  $R = B = O(1)$ , so being additive in  $F$  is asymptotically better, and would not be obtained using prior tournament trees.

The problem of obtaining SP-WF-Exit highlights the difficulty of composing recoverable locks. The issue is that a process in the main lock is composing critical sections of the node locks, which creates the problem of how recovery of the main and node locks interact. In the model of prior work [11, 14], a process crashing in the main lock's exit section attempts to re-acquire the main lock upon recovering. As a result, the process might now block in some node lock's entry section, which violates SF-WF-Exit for the main lock. We address this problem by carefully modeling RME algorithms in a way that facilitates composition (§ 2). Instead of assuming how a process participates in the algorithm (i.e., cycling through entry, CS, exit), we model the RME algorithm as an object whose *Recover* procedure informs the process where it crashed in the super-passage. This approach allows client algorithms, composing the lock, to decide how to proceed. Our model allows a process returning to lock  $x$  after crashing in the main lock to realize that it had completed an  $x$ -super-passage *and not start a new one*. Consequently, our tournament tree avoids the problems described above and satisfies SP-WF-Exit.

We present detailed pseudo code and prove all of the algorithm's properties. Due to space limits, omitted proofs appear in the full version [22, Appendix C].

```

1 STATUS: array of N status words // initially all TRY.
2 CURR_NODE: array of N nodes
3 // initially CURR_NODE[i] is the i-th leaf.

5 void Try(int pid) {
6   if STATUS[pid] = ABORT:
7     Exit(k, TRUE)
8     return FALSE
9   node = CURR_NODE[pid]
10  while STATUS[pid] ≠ CS or node ≠ root:
11    if node is the j-th child of node.parent,
12    then set k to j
13    if node.Recover(k) = TRY:
14      node.Try(k)
15    if received abort signal:
16      STATUS[pid] := ABORT
17      Exit(pid, TRUE)
18      return FALSE
19    if node = root:
20      break
21    node := node.parent
22    CURR_NODE[pid] := node
23    STATUS[pid] := CS
24    return TRUE
25 }

27 void Exit(int pid, bool aborting) {
28   if aborting = FALSE:
29     STATUS[pid] = EXIT
30     node := CURR_NODE[pid]
31     while TRUE:
32       if node is the j-th child of node,
33       then set k to be j
34       if node.Recover(k) ≠ TRY:
35         node.Exit(k, FALSE)
36       if node = LEAF:
37         break
38       node := node.child(k)
39       CURR_NODE[pid] := node
40       STATUS[pid] := TRY
41   }
42 status Recover(int pid) {
43   if STATUS[pid] = EXIT:
44     return EXIT
45   if STATUS[pid] = CS:
46     return CS
47   return TRY
48 }

```

■ **Figure 2** The Tournament Tree.

## 4.1 Algorithm Walk-Through

Figure 2 shows the pseudo code of the algorithm. Each node has immutable *parent* and *child* pointers (as mentioned before, the tree structure is static). The *parent* of root is  $\perp$ , as are all *child* pointers of a leaf node. Each process is statically assigned to a leaf based on its id (*pid*). Each node contains a  $D$ -port abortable RME lock.

Similarly to our  $W$ -port algorithm, each process  $p$  has a status word,  $STATUS[p]$ , which is used by the main lock's *Recover* procedure. Each process has a *current\_node* pointer.

In *Try*, a process walks the path from its leaf to the root, acquiring each node lock along the way (lines 10–22). In each such lock, it uses a statically assigned port, corresponding to the number of the child from which it climbed into the node. After successfully acquiring the lock at node  $x$ , process  $p$  writes  $x$  to *current\_node*[ $p$ ] (line 22). This allows  $p$  to return to  $x$  if it crashes, instead of having to start from scratch and climb the entire path again. The *Exit* flow is symmetric, with  $p$  releasing each lock along the path back to the leaf, and updating *current\_node*[ $p$ ] after each lock release (lines 31–39). In both entry and exit flows,  $p$  always execute node lock's *Recover* procedure before entering that lock's *Try* or *Exit* section. This allows  $p$  to behave correctly after crash recovery: on its way up (respectively, down) it will not execute *Enter* (respectively, *Exit*) on the same node lock twice (lines 13–14, respectively lines 34–35).

To support aborts, process  $p$  checks the abort signal after acquiring each node lock (lines 15–18). If an abort was signalled,  $p$  starts executing the main lock's exit code to descend from the current node back to its leaf, releasing the node locks it holds along the way. (Similarly to the  $W$ -port algorithm, an aborting process execute *Exit* as a subroutine; it does not formally enter the main lock's exit section). The algorithm correctly supports aborts because if an abort is signalled while  $p$  is in some node lock's *Try* execution, it is guaranteed to complete in a finite number of its own steps. Subsequently, it will execute the main lock's abort handling code in a constant number of its own steps.

## 5 Adaptive Transformation

We now present our generic adaptivity transformation, which transforms any abortable RME algorithm  $L$  whose RMR complexity depends only on  $N$  into an abortable RME algorithm whose RMR complexity also depends on the *point contention* [1, 4],  $K$ , which is the number of processes executing the algorithm concurrently with the process going through the super-passage. We show how to transform an abortable RME algorithm with passage complexity  $B < W$ , super-passage complexity  $B^*$ , and space complexity  $S$ , into an abortable RME algorithm with passage complexity  $O(\min(K, B))$ , super-passage complexity  $O(K + F)$  if  $K < B$  or  $O(B^* + F)$  otherwise, and space complexity  $O(S + N + B^2)$ .

The transformation is essentially a fast-path/slow-path construction, where the fast path is our  $W$ -port abortable RME algorithm and the slow path is the original lock  $L$ . A process  $p$  attempts to capture port  $k = 0, \dots, W - 1$  so it can use it in the fast path lock. Each such capture attempt is performed with CAS, and hence incurs an RMR. The idea is that if  $p$  fails to capture a port, then another process  $q$  succeeds. Therefore, if  $p$  fails to capture any port, the point contention is  $> W$ . In this case,  $p$  gives up and enters the slow path. The fast path and slow paths are synchronized with a 2-port abortable RME lock, again implemented with our lock (§ 3).

We present detailed pseudo code and prove all of the algorithm's properties. Due to space limits, omitted proofs appear in the full version [22, Appendix D].

```

1 void Try(int pid) {
2   if STATUS[pid] = ABORT:
3     Exit(pid, TRUE)
4     return FALSE
5   k := CURR_K[pid]
6   while k < B:
7     if K_OWNERS[k] = pid
8       or CAS(K_OWNERS[k], ⊥, pid):
9       PATH[pid] := FAST
10      if fast_path.Recover(k) = TRY
11        if fast_path.Try(k) = FALSE:
12          STATUS[pid] := ABORT
13          Exit(pid, TRUE)
14          return FALSE
15      break loop
16      k := k + 1
17      CURR_K[pid] := k
18  if PATH[pid] ≠ FAST:
19    PATH[pid] := SLOW
20    if slow_path.Recover(pid) = TRY:
21      if slow_path.Try(pid) = FALSE:
22        STATUS[pid] := ABORT
23        Exit(pid, TRUE)
24        return FALSE
25  if PATH[pid] = FAST:
26    SIDE[pid] := RIGHT
27  else // PATH[pid] = SLOW
28    SIDE[pid] := LEFT
29  if 2_rme.Recover(SIDE[pid]) = TRY:
30    if 2_rme.Try(SIDE[pid]) = FALSE:
31      STATUS[pid] := ABORT
32      Exit(pid, TRUE)
33      return FALSE
34  STATUS[pid] := CS
35 }

36 STATUS: array of N status words // initially all TRY
37 SIDE: array of N SIDE words // initially all ⊥
38 K_OWNERS: array of B pids // initially all ⊥
39 CURR_K: array of N integers // initially all 0

41 void Exit(int pid, bool aborting) {
42   if aborting = FALSE:
43     STATUS[pid] = EXIT
44   if SIDE[pid] ≠ ⊥ and
45     2_rme.Recover(SIDE[pid]) ≠ TRY
46     2_rme.exit(SIDE[pid], FALSE)
47   SIDE[pid] := ⊥

49   if PATH[pid] = FAST:
50     k := CURR_K[pid]
51     if K_OWNERS[k] = pid and
52       fast_path.Recover(k) ≠ TRY:
53       fast_path.Exit(k, FALSE)
54     K_OWNERS[k] := ⊥
55   else if PATH[pid] = SLOW:
56     if slow_path.Recover(p) ≠ TRY:
57       slow_path.Exit(p, FALSE)
58   PATH[pid] := ⊥
59   CURR_K[pid] := 0
60   STATUS[pid] := TRY
61 }

63 status Recover(int pid) {
64   if STATUS[pid] = EXIT:
65     return EXIT
66   if STATUS[pid] = CS:
67     return CS
68   return TRY
69 }

```

■ Figure 3 Adaptive Transformation.

## 5.1 Algorithm Walk-Through

Figure 3 presents the transformed algorithm’s pseudo code. The transformed algorithm uses three auxiliary abortable RME locks: a *slow\_path* lock, which is an  $N$ -process base lock being transformed into an adaptive lock, and *fast\_path* as well as  $2\_rme$  locks, both of which are instances of our  $D$ -port abortable RME (§ 3). The *fast\_path* instance uses  $D = B$  and the  $2\_rme$  instance uses  $D = 2$ .

The algorithm maintains  $K\_OWNERS$ , an array of  $B$  words (initially all  $\perp$ ) through which processes in the entry section try to capture ports to use in the fast-path lock (lines 5–17). Each process maintains a  $CURR\_K$  variable to store the next port the process attempts to capture, or its captured port (once it captures one). To capture a port, process  $p$  scans  $K\_OWNERS$ , using CAS at each slot  $k$  in an attempt to capture port  $k$ . If  $p$  captures port  $k$ , it enters the fast-path lock using that port. Overall, if  $p$  reaches slot  $k$  in  $K\_OWNERS$ , then  $k$  other processes have captured ports  $0, \dots, k - 1$ . If  $p$  reaches the end of  $K\_OWNERS$  and fails to capture a port, it enters the slow-path lock (lines 18–24). Regardless of which lock  $p$  ultimately enters, it invokes that lock’s *Recover* method first, to correctly handle the case in which  $p$  is recovering from a crash.

We use the 2-RME lock to ensure mutual exclusion between the owners of the fast-path and the slow-path. Once  $p$  acquires its lock, it enters the 2-RME lock from the right (respectively, left) if it is on the fast-path (respectively, slow-path). In the 2-RME lock,  $p$  takes on a unique right/left id, corresponding to its direction of entry. Once  $p$  acquires the 2-RME lock, it enters the CS (lines 29–33).

In the exit section,  $p$  releases the 2-RME lock (lines 44–46) and then the fast-path or slow-path lock, as appropriate (lines 49–57). After releasing the fast-path lock,  $p$  releases its port (line 54). These steps are done carefully to avoid having  $p$  return to the fast-path lock after crashing with the same port that is now being used by another process.

To handle aborts, if  $p$  receives a FALSE return value from some *Enter* execution, it executes the transformed lock’s exit code (which, as a byproduct, releases  $p$ ’s port if it has one). Subsequently,  $p$  completes the abort.

## 6 Putting It All Together & Conclusion

Let  $T$  be the RME algorithm obtained by instantiating our tournament tree (§ 4) with our  $W$ -port abortable RME algorithm (§ 3). Then  $T$ ’s RMR passage complexity is  $O(\log_W N) < W$ , super-passage complexity is  $O(\log_W N + F)$  and space complexity is  $O(NW \log_W N)$ . We can therefore apply the transformation of § 5 to  $T$ , obtaining our main result:

► **Theorem 4.** *There exists an abortable RME with  $O(\min(K, \log_W N))$  RMR passage complexity,  $O(F + \min(K, \log_W N))$  RMR super-passage complexity, and  $O(NW \log_W N)$  space complexity where  $K$  is the point contention,  $W$  is the memory word size,  $N$  is the number of processes, and  $F$  is the number of crashes in a super-passage.*

Many questions about ME properties in the context of RME remain open, and we are far from understanding how the demand for recoverability affects the possibility of obtaining other desirable properties and their cost. Can the sublogarithmic RMR bounds be improved using only primitives supported in hardware, such as FAS and FAA? It is known that a weaker crash model facilitate better bounds [12], but is relaxing the crash model necessary? What, if any, is the connection between RME and abortable mutual exclusion? Both problems involve a similar concept, of a process “disappearing” from the algorithm, and for both problems, the best known RMR bounds (assuming standard primitives) are  $O(\frac{\log N}{\log \log N})$ . Can a formal connection between these problems be established?

---

**References**

---

- 1 Y. Afek, H. Attiya, A. Fouren, G. Stupp, and D. Touitou. Long-Lived Renaming Made Adaptive. In *PODC*, 1999.
- 2 A. Alon and A. Morrison. Deterministic Abortable Mutual Exclusion with Sublogarithmic Adaptive RMR Complexity. In *PODC*, 2018.
- 3 J.H. Anderson and Y.J. Kim. An Improved Lower Bound for the Time Complexity of Mutual Exclusion. *Distributed Computing*, 15(4), 2002.
- 4 H. Attiya. Adapting to Point Contention with Long-Lived Safe Agreement. In *SIROCCO*, 2006.
- 5 H. Attiya and A. Fouren. Algorithms Adapting to Point Contention. *JACM*, 50(4), 2003.
- 6 H. Attiya, D. Hendler, and P. Woelfel. Tight RMR Lower Bounds for Mutual Exclusion and Other Problems. In *STOC*, 2008.
- 7 D.Y.C. Chan and P. Woelfel. Recoverable Mutual Exclusion with Constant Amortized RMR Complexity from Standard Primitives. In *PODC*, 2020.
- 8 T.S. Craig. Building FIFO and Priority-Queueing Spin Locks from Atomic Swap, 1993.
- 9 S. Dhoked and N. Mittal. An Adaptive Approach to Recoverable Mutual Exclusion. In *PODC*, 2020.
- 10 E.W. Dijkstra. Solution of a Problem in Concurrent Programming Control. *CACM*, 8(9), 1965.
- 11 W. Golab and D. Hendler. Recoverable Mutual Exclusion in Sub-Logarithmic Time. In *PODC*, 2017.
- 12 W. Golab and D. Hendler. Recoverable Mutual Exclusion Under System-Wide Failures. In *PODC*, 2018.
- 13 W. Golab and A. Ramaraju. Recoverable Mutual Exclusion: [Extended Abstract]. In *PODC*, 2016.
- 14 W. Golab and A. Ramaraju. Recoverable Mutual Exclusion. *Distributed Computing*, 32(6), 2019.
- 15 P. Jayanti. F-Arrays: Implementation and Applications. In *PODC*, 2002.
- 16 P. Jayanti. Adaptive and Efficient Abortable Mutual Exclusion. In *PODC*, 2003.
- 17 P. Jayanti and S. Jayanti. Constant Amortized RMR Abortable Mutex for CC and DSM. In *PODC*, 2019.
- 18 P. Jayanti, S. Jayanti, and A. Joshi. A Recoverable Mutex Algorithm with Sub-Logarithmic RMR on Both CC and DSM. In *PODC*, 2019.
- 19 P. Jayanti, S.V. Jayanti, and A. Joshi. Optimal Recoverable Mutual Exclusion Using only FASAS. In *NETYS*, 2018.
- 20 P. Jayanti and A. Joshi. Recoverable FCFS Mutual Exclusion with Wait-Free Recovery. In *DISC*, 2017.
- 21 P. Jayanti and A. Joshi. Recoverable Mutual Exclusion with Abortability. In *NETYS*, 2019.
- 22 Daniel Katzan and Adam Morrison. Recoverable, Abortable, and Adaptive Mutual Exclusion with Sublogarithmic RMR Complexity. *CoRR*, 2020. [arXiv:2011.07622](https://arxiv.org/abs/2011.07622).
- 23 J.M. Mellor-Crummey and M.L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *TOCS*, 9(1), 1991.
- 24 M. M. Michael. Hazard pointers: safe memory reclamation for lock-free objects. *TPDS*, 15(6), 2004.
- 25 M.L. Scott. Non-blocking Timeout in Scalable Queue-based Spin Locks. In *PODC*, 2002.
- 26 M.L. Scott and W.N. Scherer. Scalable Queue-based Spin Locks with Timeout. In *PPoPP*, 2001.
- 27 Gadi Taubenfeld. *Synchronization algorithms and concurrent programming*. Pearson / Prentice Hall, 2006.
- 28 J.H. Yang and J.H. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9(1), 1995.



# Optimal Resilience in Systems That Mix Shared Memory and Message Passing

Hagit Attiya 

Department of Computer Science, Technion, Haifa, Israel  
hagit@cs.technion.ac.il

Sweta Kumari

Department of Computer Science, Technion, Haifa, Israel  
sweta@cs.technion.ac.il

Noa Schiller

Department of Computer Science, Technion, Haifa, Israel  
noa.schiller@cs.technion.ac.il

---

## Abstract

We investigate the minimal number of failures that can *partition* a system where processes communicate both through shared memory and by message passing. We prove that this number precisely captures the resilience that can be achieved by algorithms that implement a variety of shared objects, like registers and atomic snapshots, and solve common tasks, like randomized consensus, approximate agreement and renaming. This has implications for the *m&m-model* of [5] and for the hybrid, cluster-based model of [28,31].

**2012 ACM Subject Classification** Theory of computation → Distributed computing models; Theory of computation → Concurrent algorithms; Computing methodologies → Distributed algorithms

**Keywords and phrases** fault resilience, m&m model, cluster-based model, randomized consensus, approximate agreement, renaming, register implementations, atomic snapshots

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2020.16

**Related Version** A full version of the paper is available at <http://arxiv.org/abs/2012.10846>.

**Funding** This research was supported by ISF grant 380/18.

**Acknowledgements** We thank Vassos Hadzilacos, Xing Hu, Sam Toueg and the anonymous reviewers for helpful comments.

## 1 Introduction

Some distributed systems combine more than one mode of communication among processes, allowing them both to send messages among themselves and to access shared memory. Examples include recent technologies such as *remote direct memory access (RDMA)* [2–4], *disaggregated memory* [30], and *Gen-Z* [1]. In these technologies, the crash of a process does not prevent access to its shared memory by other processes. Under these technologies, it is infeasible to share memory among a large set of processes, so memories are shared by smaller, strict subsets of processes.

Systems mixing shared memory and message passing offer a major opportunity since information stored in shared variables remains available even after the failure of the process who stored it. Mixed systems are expected to withstand more process failures than pure message-passing systems, as captured by the *resilience* of a problem – the maximal number of failures that an algorithm solving this problem can tolerate. This is particularly the case in an *asynchronous* system. At one extreme, when all processes can access the same shared memory, many problems can be solved even when all processes but one fail. Such



© Hagit Attiya, Sweta Kumari, and Noa Schiller;  
licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 16; pp. 16:1–16:16

Leibniz International Proceedings in Informatics



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

*wait-free* algorithms exist for implementing shared objects and solving tasks like randomized consensus, approximate agreement and renaming. At the other extreme, when processes only communicate by message passing, the same problems require that at least a majority of processes do not fail [7,8,18]. Thus, typically, shared-memory systems are  $(n-1)$ -resilient, and pure message-passing systems are  $\lfloor (n-1)/2 \rfloor$ -resilient, where  $n$  is the number of processes.

The resilience in systems that mix shared memory and message passing falls in the intermediate range, between  $\lfloor (n-1)/2 \rfloor$  and  $n-1$ . It is, however, challenging to solve specific problems with the best-possible resilience in a particular system organization: the algorithm has to coordinate between non-disjoint sets of processes that have access to different regions of the shared memory. On the other hand, bounding the resilience requires to take into account the fact that processes might be able to communicate *indirectly* through shared memory accesses of third-party processes.

This paper explores the *optimal* resilience in systems that provide message-passing support between all pairs of processes, and access to shared memory between subsets of processes. We do this by studying the minimal number of failures that can *partition* the system, depending on its structure, i.e., how processes share memory with each other. We show that the partitioning number exactly characterizes the resilience, that is, a host of problems can be solved in the presence of  $< f$  crash failures, if and only if  $f$  is the minimal number of failures that partition the system.

A key step is to focus on the implementation of a *single-writer multi-reader register* shared among all processes, in the presence of  $f$  crash failures. A read or a write operation takes  $O(1)$  round-trips, and requires  $O(n)$  messages. Armed with this implementation, well-known shared-memory algorithms can be employed to implement other shared objects, like *multi-writer multi-reader registers* and *atomic snapshots*, or to solve fundamental problems, such as *randomized consensus*, *approximate agreement* and *renaming*. Because the register implementation is efficient, these algorithms inherit the good efficiency of the best-known shared-memory algorithm for each of these problems.

Going through a register simulation, instead of solving consensus, approximate agreement or renaming from scratch, does not deteriorate their resilience. One of our key contributions is to show that the resilience achieved in this way is optimal, by proving that these problems cannot be solved in the presence of  $f$  crash failures, if  $f$  failures can partition the system.

We consider memories with access restrictions and model mixed systems by stating which processes can read from or write to each memory. (Note that every pair of processes can communicate using messages.) Based on this concept, we define  $f_{opt}$  to be the largest number of failures that do not partition the system. We prove that  $f$ -resilient registers and snapshot implementations, and  $f$ -resilient solutions to randomized consensus, approximate agreement and renaming, exist if and only if  $f \leq f_{opt}$ .

One example of a mixed model is the *message-and-memory* model [5], in short, the *m&m* model. In the *general* m&m model [5], the shared-memory connections are defined by (not necessarily disjoint) subsets of processes, where each subset of processes share a memory. Most of their results, however, are for the *uniform* m&m model, where shared-memory connections can be induced by an undirected graph, whose vertices are the processes. Each process has an associated shared memory that can be accessed by all its neighbors in the *shared-memory graph* (see Section 5). They present bounds on the resilience for solving randomized consensus in the uniform model. Their algorithm is based on Ben-Or's exponential algorithm for the pure message-passing model [15]. The algorithm terminates if the nonfaulty processes and their neighbors (in the shared-memory graph) are a majority of the processes. They also prove an upper bound on the number of failures a randomized consensus algorithm can

tolerate in the uniform m&m model. We show that in the uniform m&m model, this bound is equivalent to the partitioning bound ( $f_{opt}$ ) proved in our paper (Theorem 17 in Section 5). We further show that this bound *does not match the resilience of their algorithm*, whose resilience is strictly smaller than  $f_{opt}$ , for some shared-memory graphs.

In the special case where the shared memory *has no access restrictions*, our model is dual to the general m&m model, i.e., it captures the same systems as the general m&m model. However, rather than listing which processes can access a memory, we consider the flipped view: we consider for each process, the memories it can access. We believe this makes it easier to obtain some extensions, for example, for memories with access restrictions.

Hadzilacos, Hu and Toueg [23] present an implementation of a SWMR register in the general m&m model. The resilience of their algorithm is shown to match the maximum resilience of an SWMR register implementation in the m&m model. Our results for register implementations are adaptations of their results. For the general m&m model specified by the set of process subsets  $L$ , they define a parameter  $f_L$  and show that it is the maximum number of failures tolerated by an algorithm implementing a SWMR register [23] or solving randomized consensus [24]. For memories without access restrictions,  $f_L$  is equal to  $f_{opt}$ . Their randomized consensus algorithm is based on the simple algorithm of [6] and inherits its exponential expected step complexity.

Another example of a model that mixes shared memory and message passing is the hybrid model of [28,31]. In this model, which we call *cluster-based*, processes are partitioned into disjoint *clusters*, each with an associated shared memory; all processes in the cluster (and only them) can read from and write to this shared memory. Two randomized consensus algorithms are presented for the cluster-based model [31]. Their resilience is stated as an operational property of executions: the algorithm terminates if the clusters of responsive processes contain a majority of the processes. We prove (Lemma 19 in Section 6) that the optimal resilience we state in a closed form for the cluster-based model is equal to their operational property.

Our model is general and captures all these models within a single framework, by precisely specifying the shared-memory layout. The tight bounds in this general model provide the exact resilience of any system that mix shared memory and message passing.

## 2 Modelling Systems that Mix Shared Memory and Message Passing

We consider  $n$  asynchronous processes  $p_1, \dots, p_n$ , which communicate with each other by sending and receiving messages, over a complete communication network of asynchronous reliable links. In addition, there are  $m$  shared memories  $M = \{\mu_1, \dots, \mu_m\}$ , which can be accessed by subsets of the processes. A memory  $\mu \in M$  has access restrictions, where  $R_\mu$  denotes all the processes that can read from the memory and  $W_\mu$  denotes all the processes that can write to the memory. The set of memories a process  $p$  can read from is denoted  $R_p$ , i.e.,  $R_p = \{\mu \in M : p \in R_\mu\}$ . The set of memories  $p$  can write to is denoted  $W_p$ , i.e.,  $W_p = \{\mu \in M : p \in W_\mu\}$ . We assume the network allows nodes to send the same message to all nodes; message delivery is FIFO. A process  $p$  can *crash*, in which case it stops taking steps; messages sent by a crashed process may not be delivered at their recipients. We assume that the shared memory does not fail, as done in prior work [5, 23, 28, 31].

A *configuration*  $C$  is a tuple with a state for each process, a value for each shared register, and a set of messages in transit (sent but not received) between any pair of processes. A *schedule* is a sequence of process identifiers. For a set of processes  $P$ , a schedule is *P-free* if no process from  $P$  appears in the schedule; a schedule is *P-only* if only processes from  $P$  appear in the schedule. An *execution*  $\alpha$  is an alternating sequence of configurations and

*events*, where each event is a step by a single process that takes the system from the preceding configuration to the following configuration. In a step, a process either accesses the shared memory (read or write) or receives and sends messages. Additionally, a step may involve the invocation of a higher-level operation. A schedule is associated with the execution in a natural way; this induces notions of  $P$ -free and  $P$ -only executions.

If there is a shared memory  $\mu \in M$  that  $p$  can read from and  $q$  can write to, then we denote  $p \rightarrow q$ . If  $p \rightarrow q$  and  $q \rightarrow p$ , then we denote  $p \leftrightarrow q$ . Since a process can read what it writes to its local memory, this relation is *reflexive*, i.e., for every process  $p$ ,  $p \rightarrow p$ . Let  $P$  and  $Q$  be two sets of processes. Denote  $P \rightarrow Q$  if some process  $p \in P$  can read what a process  $q \in Q$  writes, i.e.,  $p \rightarrow q$ . If  $P \rightarrow Q$  and  $Q \rightarrow P$ , then we denote  $P \leftrightarrow Q$ .

► **Definition 1.** A system is  $f$ -partitionable if there are two sets of processes  $P$  and  $Q$ , both of size  $n - f$ , such that  $P \not\leftrightarrow Q$ . Namely, the failure of  $f$  processes can partition (disconnect) two sets of  $n - f$  processes. Denote by  $f_{opt}$  the largest integer  $f$  such that  $P \leftrightarrow Q$ , for every pair of sets of processes  $P$  and  $Q$ , each of size  $n - f$ .

Clearly, a system is  $f$ -partitionable if and only if  $f > f_{opt}$ . Note that  $f_{opt} \geq \lfloor (n - 1)/2 \rfloor$ . In the pure message-passing model,  $p \rightarrow q$  if and only if  $p = q$ ; hence,  $f_{opt} = \lfloor (n - 1)/2 \rfloor$ .

The special case of shared memory *without access restrictions* is when for every memory  $\mu \in M$ ,  $R_\mu = W_\mu$ , and all processes that can read from a memory can also write to it. In this case, the  $\rightarrow$  relation is *symmetric*, i.e., for every pair of processes  $p$  and  $q$ , if  $p \rightarrow q$  then  $q \rightarrow p$ . Therefore, for every two processes  $p$  and  $q$ ,  $p \leftrightarrow q$ . Later, we discuss two models without access restrictions, the m&m model and the cluster-based model.

For a set of processes  $P$ ,  $\vec{P}$  are the processes that some process in  $P$  can read what they write to the shared memory, i.e.,  $\vec{P} = \{q : \exists p \in P, p \rightarrow q\}$ .  $f_{maj}$  is the largest integer  $f$  such that for every set  $P$  of  $n - f$  processes,  $|\vec{P}| > \lfloor n/2 \rfloor$ . That is,  $f_{maj}$  is the largest number of failures that still allows the remaining (nonfaulty) processes to communicate with a majority of the processes. It is simple to see that  $f_{opt} \leq f_{maj}$ . The converse direction does not necessarily hold, as discussed for the m&m model and the cluster-based model.

### 3 Necessary and Sufficient Condition for Implementing a Register

This section shows that a register can be implemented in the presence of  $f$  failures, if and only if the system is not  $f$ -partitionable, that is,  $f \leq f_{opt}$ . This is an adaptation of the register implementation of [23] in the m&m model. A *single-writer multi-reader* (SWMR) register  $R$  can be written by a single writer process  $w$ , using a procedure WRITE, and can be read by all processes  $p_1, \dots, p_n$ , using a procedure READ. A register is *atomic* [29] if any execution of READ and WRITE operations can be linearized [27]. This means that there is a total order of all completed operations and some incomplete operations, that respects the real-time order of non-overlapping operations, in which each READ operation returns the value of the last preceding WRITE operation (or the initial value of the register, if there is no such WRITE).

The algorithm appears in Algorithm 1; for simplicity of presentation, a process sends each message also to itself and responds with the appropriate response. All the message communication between the processes is done in `msg_exchange()`, where we simply send a message and wait for  $n - f$  acknowledgement. This modular approach allows us to replace the communication pattern according to the specific shared-memory layout. For example, Section 6 shows that in the cluster-based model this communication pattern can be changed to wait for less than  $n - f$  processes.

■ **Algorithm 1** Atomic SWMR register implementation ( $w$  is the single writer).

---

**Local Variables:**

$w\text{-sqno}$ : int, initially 0 ▷ write sequence number  
 $r\text{-sqno}$ : int, initially 0 ▷ read sequence number  
 $last\text{-sqno}$ : int, initially 0 ▷ last write sequence number observed  
 $counter$ : int, initially 0 ▷ number of replies/acks received so far

**Shared Variables:** for every process  $p$  and every  $\mu \in W_p$ :

$R_\mu[p]$ :  $\langle \text{int}, \text{int} \rangle$ , initially  $\langle 0, v_0 \rangle$  ▷ writable by  $p$  and readable by all processes that can read from  $\mu$ , i.e., all the processes in  $R_\mu$

---

**WRITE( $v$ ) – Code for the writer  $w$ :**

1:  $w\text{-sqno} = w\text{-sqno} + 1$  ▷ increment the write sequence number  
2:  $acks = \text{msg\_exchange}\langle W, w\text{-sqno}, v \rangle$   
3: return

**Code for any process  $p$ :**

4: Upon receipt of a  $\langle W/WB, sqno, v \rangle$  message from process  $w/q$ :  
5: **if** ( $sqno > last\text{-sqno}$ ) **then**  
6:      $last\text{-sqno} = sqno$   
7:     **for each**  $\mu \in W_p$  **do** ▷ write value and sequence number to every register  $p$  can write to  
8:          $R_\mu[p] = \langle sqno, v \rangle$   
9:     send  $\langle \text{Ack-W/Ack-WB}, sqno \rangle$  to process  $w/q$

**READ() – Code for the reader  $q$ :**

10:  $r\text{-sqno} = r\text{-sqno} + 1$  ▷ increment the read sequence number  
11:  $set\_of\_tuples = \text{msg\_exchange}\langle R, r\text{-sqno}, \perp \rangle$   
12:  $\langle seq, val \rangle = \max(set\_of\_tuples)$  ▷ maximum  $\langle seq, val \rangle$   
13:  $acks = \text{msg\_exchange}\langle WB, seq, val \rangle$  ▷ write back  
14: return  $val$

**Code for any process  $p$ :**

15: Upon receipt of a  $\langle R, r\text{-sqno}, - \rangle$  message from process  $q$ :  
16:  $\langle w\text{-seq}, w\text{-val} \rangle = \max\{\langle seq, val \rangle : \mu \in R_p \cap W_q \text{ and } R_\mu[q] = \langle seq, val \rangle\}$  ▷ find  $val$  with maximum  $seq$   
17: send  $\langle \text{Ack-R}, r\text{-sqno}, \langle w\text{-seq}, w\text{-val} \rangle \rangle$  to process  $q$

**msg\_exchange( $m, seq, val$ ): returns set of responses**

18: send  $\langle m, seq, val \rangle$  to all processes  
19:  $responses = \emptyset$   
20: **repeat**  
21:     wait to receive a message  $m$  of the form  $\langle \text{Ack-}m, seq, - \rangle$   
22:      $counter = counter + 1$   
23:      $responses = responses \cup \{m\}$   
24: **until**  $counter \geq n - f$   
25: return( $responses$ )

---

For each process  $p$  and memory  $\mu \in W_p$  there is a shared SWMR register  $R_\mu[p]$ , writable by  $p$  and readable by every process that can read from  $\mu$ , i.e., every process in  $R_\mu$ . In  $\text{WRITE}(v)$ , the writer  $w$  increments its local write sequence number  $w\text{-sqno}$  and calls  $\text{msg\_exchange}()$ .

This procedure sends a message of type W with value  $v$  and  $w\text{-sqno}$  to all processes. On receiving a write message from  $w$ ,  $p$  checks if the write value is more up-to-date than the last value it has observed, by checking if  $w\text{-sqno}$  is larger than  $\text{last-sqno}$ . If so,  $p$  updates  $\text{last-sqno}$  to be  $w\text{-sqno}$  and writes the value  $v$  and sequence number  $w\text{-sqno}$  to all the registers it can write to. When done, the process sends an acknowledgment to the writer  $w$ . Once  $w$  receives  $n - f$  acknowledgments, it returns successfully.

In READ, a reader process  $q$  increments its local read sequence number  $r\text{-sqno}$  and calls `msg_exchange()`. This procedure sends a message of type R and  $r\text{-sqno}$  to all processes. On receiving a read message from  $q$ , a process  $p$  reads all the registers it can read and finds the maximum sequence number and value stored in them and sends this pair to the reader  $q$ . Once  $q$  receives  $n - f$  acknowledgments, it finds the value  $\text{val}$  with maximum sequence number  $\text{seq}$  among the responses (i.e., it selects the most up-to-date value). Then,  $q$  calls `msg_exchange()`, with message type WB (write back) and value  $\text{val}$  and  $\text{seq}$  to update other readers. On receiving a write back message from  $q$ , each process  $p$  handles WB like W message, checking if  $w\text{-sqno}$  is larger than  $\text{last-sqno}$  and if so updating  $\text{last-sqno}$  and all the registers it can write to. When done, the process sends an acknowledgment to  $q$ . Once  $q$  receives  $n - f$  acknowledgments, it returns  $\text{val}$  successfully.

The communication complexities of read and write operations are dominated by the cost of a `msg_exchange()`, invoked once in a write and twice in a read. This procedure takes one round-trip and  $O(n)$  messages, like the algorithm for the pure message-passing model [7]. The number of shared SWMR registers depends on the shared-memory topology and is  $\rho = \sum_{\text{process } p} |W_p|$ , as every process has a single register in each memory it can write to. The number of accesses to the shared memory is  $\sigma = \sum_{\text{process } p} \sum_{\mu \in R_p} |W_\mu|$ , as every process reads all the registers it can read from. Note that  $\sigma \leq n\rho$ .

The only statement that could prevent the completion of a WRITE or a READ is waiting for  $n - f$  responses (Line 24). Since at most  $f$  processes may crash, the wait statement eventually completes, implying that a WRITE or READ invoked by a process that does not crash completes.

► **Lemma 2.** *Let  $t_2$  be the largest sequence number returned in a read `msg_exchange` by reader  $p_j$ , and assume that the `msg_exchange` starts after the completion of a write `msg_exchange`, either by the writer  $w$  or in a write back by reader  $p_i$ , with sequence number  $t_1$ , then,  $t_1 \leq t_2$ .*

We explicitly order all completed reads and all invoked writes (even if they are incomplete). Note that values written by the writer  $w$  have distinct write sequence numbers, and are different from the initial value of the register, denoted  $v_0$ ; the value of the  $k^{\text{th}}$  write operation is denoted  $v_k$ ,  $k \geq 1$ . Writes are ordered by the order they are invoked by process  $w$ ; if the last write is incomplete, we place this write at the end. Since only one process invokes write, this ordering is well-defined and furthermore, the values written appear in the order  $v_1, v_2, \dots$

Next, we consider reads in the order they complete; note that this means that non-overlapping operations are considered in their order in the execution. A read that returns the value  $v_{k-1}$ ,  $k \geq 0$ , is placed before the  $k$ -th write in the ordering, if this write exists, and at the end of the ordering, otherwise. For  $k = 0$ , this means that the read is placed before the first write, which may be at the end of the order, if there is no write.

Lemma 2 implies that this order respects the real-time order of non-overlapping operations.

► **Theorem 3.** *If a system is not  $f$ -partitionable then Algorithm 1 implements an atomic SWMR register, in the presence of  $f$  failures.*

The impossibility proof holds even if only *regular* register [29] is implemented. In a regular register, a read should return the value of a WRITE operation that either overlaps it, or immediately precedes it. The proof is similar to the one in [23], where they show that a SWMR register cannot be implemented in the m&m model if more than  $f_L$  processes may fail.

► **Theorem 4.** *If a system is  $f$ -partitionable then there is no implementation of a regular SWMR register in the presence of  $f$  failures.*

## 4 Solving Other Problems in Non-Partitionable Systems

### 4.1 Constructing Other Read/Write Registers

The atomic SWMR register presented in the previous section can be used as a basic building block for implementing other shared-memory objects. Recall that if a system is not  $f$ -partitionable (i.e.,  $f \leq f_{opt}$ ), a SWMR register can be implemented so that each operation takes  $O(1)$  time,  $O(n)$  messages,  $O(\rho)$  SWMR shared-memory registers and  $O(\sigma)$  SWMR shared-memory accesses. Given a shared-memory algorithm that uses  $O(r)$  SWMR registers and has  $O(s)$  step complexity, it can be simulated with  $O(s)$  round-trips,  $O(ns)$  messages, and  $O(\sigma s)$  shared-memory accesses. The simulation requires  $O(\rho r)$  SWMR shared-memory registers. (Recall that  $\rho = \sum_{\text{process } p} |W_p|$  and  $\sigma = \sum_{\text{process } p} \sum_{\mu \in R_p} |W_\mu|$ .)

An atomic *multi-writer multi-reader* (MWMR) register can be built from atomic SWMR registers [33]; each read or write requires  $O(n)$  round-trips,  $O(n^2)$  messages,  $O(\rho n)$  SWMR shared registers and  $O(\alpha n)$  shared-memory accesses.

Atomic snapshots can also be implemented using SWMR registers [13]; each scan or update takes  $O(n \log n)$  round-trips,  $O(n^2 \log n)$  messages,  $O(\rho n)$  SWMR shared registers and  $O(\sigma n \log n)$  shared-memory accesses.

### 4.2 Batching

A simple optimization is *batching* of read requests, namely reading the registers of several processes simultaneously. Batching is useful when each process replicates a register for each other process – not for just one writer. A process  $p$  can send read requests for all these registers together, instead of sending  $n$  separate read requests (for the registers of all processes), one after the other. When a process  $q$  receives the batched request from  $p$ , it replies with a vector containing the values of all registers in a single message, rather than sending them separately. Process  $p$  waits for vectors from  $n - f$  processes, and picks from them the latest value for each other process. Finally, the reader does a write-back of this vector.

Batching reduces the number of round-trips and messages, and shared-memory registers and accesses, but increases the size of messages and registers. With batching, an operation on a MWMR register requires  $O(1)$  round-trips,  $O(n)$  messages,  $O(\rho)$  SWMR shared registers and  $O(\alpha)$  shared-memory accesses, when each process saves all the writers values in a single SWMR register. Batching can also be applied to atomic snapshots, so that each scan or update takes  $O(\log n)$  round-trips,  $O(n \log n)$  messages,  $O(\rho)$  SWMR shared registers and  $O(\sigma \log n)$  shared-memory accesses.

Batching provides a *regular collect*, as defined in [11]. Regular collects can be used in the following building block, where a process repeatedly call collect, and returns a vector of values if it has received it *twice* (in two consecutive collects). Two vectors are the same if they contain the same sequence numbers in each component. Process  $p$  can write the value  $v$  using



procedure  $\text{WRITE}_p(v)$ , and repeatedly double collect all the processes current values using the procedure  $\text{BUILDINGBLOCK}()$ . An invocation of  $\text{BUILDINGBLOCK}()$  returns a vector  $V$  with  $n$  components, one for each process. Each component contains a pair of a value with a sequence number. For every process  $p_i$ ,  $V[i]$  is the entry in the vector corresponding to  $p_i$ 's value. A vector  $V_1$  *precedes* a vector  $V_2$  if the sequence number of each component of  $V_1$  is smaller than or equal to the corresponding component of  $V_2$ . Although the writes are not atomic, it can be shown that if  $V_1$  and  $V_2$  are vectors returned by two pairs of successful double collects then either  $V_1$  precedes  $V_2$  or  $V_2$  precedes  $V_1$ .

This building block may not terminate (even if the system is not  $f$ -partitionable), due to continuous writes. However, if two consecutive collects are not equal then some sequence number was incremented, i.e., a write by some process is in progress.

### 4.3 Consensus

In the *consensus* problem, a process starts with an input value and decides on an output value, so that all processes decide on the same value (*agreement*), which is the input value of some process (*validity*). With a standard *termination* requirement, it is well known that consensus cannot be solved in an asynchronous system [21]. This result holds whether processes communicate through shared memory or by message passing, and even if only a single process fails. However, consensus can be solved if the termination condition is weakened, either to be required only with high probability (*randomized consensus*), or to hold when it is possible to eventually detect failures (using a *failure detector*), or to happen only under fortunate situations.

There are numerous shared-memory randomized consensus algorithms, which rely on read / write registers, or objects constructed out of them. Using these algorithms together with linearizable register implementations is not obvious since linearizability does not preserve *hyperproperties* [9, 22]. It has been shown [24] that the ABD register implementation [7] is not *strongly linearizable* [22]. This extends to the mixed-model register implementations, as ABD is a special case of them.

Hadzilacos et al. [25] have proved that the simple randomized consensus algorithm of [6] works correctly with *regular* registers, and used it to obtain consensus in m&m systems [24]. Their algorithm inherits exponential complexity from the simple algorithm of [6], which employs independent coin flips by the processes.

Here, we explain how to use  $\text{BUILDINGBLOCK}()$  to emulate the weak shared coin of [6], following [14]. This holds with  $f$  failures, if the system is not  $f$ -partitionable.

In Algorithm 2, a process flips a coin using a local function  $\text{flip}()$ , which returns the value 1 or -1, each with probably 1/2. Invoking  $\text{flip}()$  is a single atomic step. After each flip, a process writes its outcome in an individual cumulative sum. Then it calls  $\text{BUILDINGBLOCK}()$  to obtain a vector  $V$  with the individual cumulative sums of all processes. (We assume that the initial value in each component is 0.) The process then checks the absolute value of the total sum of the individual cumulative sums, denoted  $\text{sum}(V)$ . If it is at least  $c \cdot n$  for some constant  $c > 1$ , then the process returns its sign.

Intuitively, the only way the adversary can create disagreement on the outcome of the shared coin is by preventing as many processors as possible to move the counter in the unwanted direction. We will show that the adversary cannot “hide” more than  $n - 1$  coin flips. (This was originally proved when processes use atomic writes [6]; here, we show it holds even when writes are not atomic.) Therefore, after the cumulative sum is big or small enough the adversary can no longer affect the outcome of the shared coin, and cannot prevent the processes from terminating.

■ **Algorithm 2** Weak shared coin [6].

---

**Local Variables:**

*my-counter*: int, initially 0

*V*: vector of size  $n$ , with all entries initially 0

---

COIN() – Code for process  $p$ :

```

1: while true do
2:   my-counter = my-counter + flip()
3:   WRITEp(my-counter)
4:   V = BUILDINGBLOCK()
5:   if sum(V) ≥ c · n then return 1
6:   else if sum(V) ≤ -c · n then return -1

```

---

Let  $H$  and  $T$  be the number of 1 and -1 (respectively) flipped by all processes at some point in the execution. These numbers are well-defined since the local coin flips are atomic.

► **Lemma 5.** *If  $H - T < -(c + 1) \cdot n$  (respectively,  $H - T > (c + 1) \cdot n$ ) at some point in the execution, then a process that invokes BUILDINGBLOCK() after this point returns -1 (respectively, 1).*

**Proof.** (Sketch) We consider the first case; the other case is symmetric. Consider the set of processes that invoked BUILDINGBLOCK() after the point in the execution when  $H - T < -(c + 1) \cdot n$ , in the order their BUILDINGBLOCK() returns. Let  $p_{j_i}$ ,  $i \geq 1$ , be the  $i$ th process in this order, and let  $V_i$  be the vector returned by its BUILDINGBLOCK(). We prove, by induction on  $i$ , that  $\text{sum}(V_i) \leq -c \cdot n$ , and hence,  $p_{j_i}$  returns -1.

In the base case,  $i = 1$ . Since a process invokes BUILDINGBLOCK() after every write, there can be at most  $n$  writes (either pending or finished) after the point  $H - T < -(c + 1) \cdot n$  and the return of BUILDINGBLOCK() by  $p_{j_1}$ . Therefore,  $\text{sum}(V_1) < -c \cdot n$ , and  $p_{j_1}$  decides -1 in Line 6.

Inductive step: Assume that for  $i > 1$ , processes  $p_{j_1}, \dots, p_{j_{i-1}}$  decide after their BUILDINGBLOCK() invocation returns. Therefore, there are no additional writes in the execution, and  $p_{j_i}$  will observe at most  $n$  additional values from  $H - T$  and will return -1. ◀

► **Lemma 6.** *If process  $p$  returns 1 (respectively, -1) from the shared coin, then  $H - T > (c - 1) \cdot n$  (respectively,  $H - T < -(c - 1) \cdot n$ ) at some point during its last call to BUILDINGBLOCK().*

**Proof.** (Sketch) We consider the first case; the other case is symmetric. Consider the last pair of collects in the last BUILDINGBLOCK() invocation before process  $p$  returns, and assume they return a vector  $V$ . Assume  $p$  misses a write by some process  $q$  that overlaps the first collect, i.e., the sequence number of this write is smaller than the corresponding sequence number in  $V$ . Then  $q$ 's write overlaps  $p$ 's first collect, and it returns after the second collect starts. (Otherwise, the regularity of collect implies that the second collect returns this write by  $q$ , or a later one, contradicting the fact it is equal to the first collect.) Therefore, each process has at most one write that overlaps the first collect and can be missed by the first collect. So, the sum of  $V$  differs by at most  $n - 1$  values from  $H - T$  at the point when the first collect completes. Since  $p$  returns 1,  $\text{sum}(V) \geq c \cdot n$ , and it holds that  $H - T > (c - 1) \cdot n$  when the first collect completes. ◀

The next lemma can be proved along the lines of [6, Theorem 17], using the fact (see proof of Lemma 5) that there are at most  $n$  additional writes after  $H - T$  drops below  $-(c + 1) \cdot n$ .

## 16:10 Resilience of Systems That Mix Shared Memory and Message Passing

► **Lemma 7.** *The adversary can force the weak shared coin procedure of a process to return 1 (respectively, -1) with probability at most  $(c + 1)/2c$ .*

It follows that the adversary can force the processes to disagree with probability at most  $(c - 1)/2c$ . The next theorem has the same proof as in [6].

► **Theorem 8.** *For a constant  $c > 1$ , the expected number of coin flips in an execution of the weak shared coin is  $O(n^2)$ .*

Since the expected number of coin flips is  $O(n^2)$ , the expected number of write and building block invocations is also  $O(n^2)$ . The total number of collect operations in these building block invocations for all the processes is  $O(n^3)$  in expectation, this is because a double collect fails only when another coin is written. Therefore, the complexity of the weak shared coin is  $O(n^3)$  round-trips,  $O(n^4)$  messages,  $O(\rho)$  registers and  $O(\alpha n^3)$  shared-memory accesses. Plugging the weak shared coin in the overall algorithm of [6], proved to be correct by [25], yields a randomized consensus algorithm with the same expected complexities as the weak shared coin.

Next, we prove that randomized consensus cannot be solved in a partitionable system, by considering the more general problem of *non-deterministic  $f$ -terminating* consensus, an extension of *nondeterministic solo termination* [20]. This variant of consensus has the usual validity and agreement properties, with the following termination property:

**Non-deterministic  $f$ -termination:** For every configuration  $C$ , process  $p$  and set  $F$  of at most  $f$  processes, such that  $p \notin F$ , there is an  $F$ -free execution in which process  $p$  terminates.

► **Theorem 9.** *If a system is  $f$ -partitionable then non-deterministic  $f$ -terminating consensus is unsolvable.*

**Proof.** Assume, by way of contradiction, that there is a non-deterministic  $f$ -terminating consensus algorithm. Since the system is  $f$ -partitionable, there are two disjoint sets of processes  $P$  and  $P'$ , each of size  $n - f$ , such that  $P' \not\rightarrow P$ . Therefore, there are no two processes  $p \in P$  and  $p' \in P'$  so that  $p'$  can read from a memory and  $p$  can write to that same memory. Let  $Q$  be the processes not in  $P \cup P'$ . Since  $|P|, |P'| = n - f$ , it follows that  $|P \cup Q| = |P' \cup Q| = f$ .

To prove the theorem, we construct three executions. Consider an initial configuration, in which all processes in  $P$  have initial value 0. Since  $|P' \cup Q| = f$ , non-deterministic  $f$ -termination implies there is a  $(P' \cup Q)$ -free execution, in which some process  $p \in P$  terminates, say by time  $t_1$ . Call this execution  $\alpha_1$ , and note that only processes in  $P$  take steps in  $\alpha_1$ . By validity,  $p$  decides 0.

In a similar manner, we can get a  $(P \cup Q)$ -free execution,  $\alpha_2$ , in which initial values of all the processes in  $P'$  are 1, and by non-deterministic  $f$ -termination, some process  $p' \in P'$  decides on 1, say by time  $t_2$ . Note that only processes in  $P'$  take steps in  $\alpha_2$ .

Finally, the third execution  $\alpha_3$  combines  $\alpha_1$  and  $\alpha_2$ . The initial value of processes in  $P$  is 0, and the initial value of processes in  $P'$  is 1. Processes in  $Q$  have arbitrary initial values, and they take no steps in  $\alpha_3$ . The execution is identical to  $\alpha_1$  from time 0 until time  $t_1$ , and to  $\alpha_2$  from this time until time  $t_1 + t_2$ . All messages sent between processes in  $P$  and processes in  $P'$  are delivered after time  $t_1 + t_2$ . Since processes in  $P'$  do not take steps in  $\alpha_3$  until time  $t_1$ , all processes in  $P$  decides 0, as in  $\alpha_1$ . Processes in  $P'$  cannot receive messages from processes in  $P$  or read what processes in  $P$  write to the shared memory, therefore all processes in  $P'$  decides 1, as in execution  $\alpha_2$ , violating the agreement property. ◀

## 4.4 Approximate Agreement

In the *approximate agreement* problem with parameter  $\epsilon > 0$ , all processes start with a real-valued input and must decide on an output value, so any two decision values are in distance at most  $\epsilon$  from each other (*agreement*), and any decision value is in the range of all initial values (*validity*).

There is a wait-free algorithm for the approximate agreement problem in the shared-memory model, which uses only SWMR registers [12]. This algorithm can be simulated if the system is not  $f$ -partitionable, and at most  $f$  processes fail. Similarly to randomized consensus, it can be shown that this problem is unsolvable in partitionable systems.

► **Theorem 10.** *If a system is  $f$ -partitionable then approximate agreement is unsolvable in the presence of  $f$  failures.*

## 4.5 Renaming

In the  $M$ -*renaming* problem, processes start with unique *original* names from a large namespace  $\{1, \dots, N\}$ , and the processes pick distinct *new* names from a smaller namespace  $\{1, \dots, M\}$  ( $M < N$ ). To avoid a trivial solution, in which a process  $p_i$  picks its index  $i$  as the new name, we require *anonymity*: a process  $p_i$  with original name  $m$  performs the same as process  $p_j$  with original name  $m$ .

Employing the SWMR register simulation in a  $(2n - 1)$ -renaming algorithm [10] yields an algorithm that requires  $O(n \log n)$  round-trips,  $O(n^2 \log n)$  messages,  $O(\rho n^4)$  shared registers and  $O(\sigma n \log n)$  shared-memory accesses. The number of registers can reduce to  $O(\rho)$ , at the cost of increasing their size.

This algorithm assumes that the system is not  $f$ -partitionable and at most  $f$  processes fail. The next theorem shows that this is a necessary condition.

► **Theorem 11.** *If a system is  $f$ -partitionable then renaming is unsolvable in the presence of  $f$  failures.*

**Proof.** Assume, by way of contradiction, that there is a renaming algorithm. Since the system is  $f$ -partitionable, there are two disjoint sets of processes  $P$  and  $P'$ , each of size  $n - f$ , such that  $P' \not\rightarrow P$ . Denote  $P = \{p_{i_1}, \dots, p_{i_{n-f}}\}$  and  $P' = \{p'_{i_1}, \dots, p'_{i_{n-f}}\}$ . Let  $Q$  be the set of processes not in  $P \cup P'$ . Since  $|P|, |P'| = n - f$ , we have that  $|P \cup Q| = |P' \cup Q| = f$ .

Given a vector  $I$  of  $n - f$  original names, denote by  $\alpha(I, P)$  the  $P$ -only execution in which processes in  $P$  have original names  $I$ : processes in  $(P' \cup Q)$  crash and take no step, and processes in  $P$  are scheduled in round-robin. Since at most  $f$  processes fail in  $\alpha(I, P)$ , eventually all processes in  $P$  pick distinct new names, say by time  $t(I)$ . Note that by anonymity, the same names are picked in the execution  $\alpha(I, P')$ , in which  $p'_{i_j}$  starts with the same original name as  $p_{i_j}$  and takes analogous steps.

Consider  $\alpha(I_i, P)$ , for any possible set of original names. The original name space can be picked to be big enough to ensure that for two *disjoint* name assignments,  $I_1$  and  $I_2$ , some process  $p_{i_j} \in P$  decides the same new name  $r$  in the executions  $\alpha(I_1, P)$  and  $\alpha(I_2, P)$ .

Denote  $\alpha_1 = \alpha(I_1, P)$  and  $\alpha_2 = \alpha(I_2, P')$ , namely, the execution in which processes in  $P'$  replace the corresponding processes from  $P$ . The anonymity assumption ensures that  $p'_{i_j}$  decides on  $r$ , just as  $p_{i_j}$  decides on  $r$  in  $\alpha(I_1, P)$  and  $\alpha(I_2, P)$ .

The execution  $\alpha_3$  combines  $\alpha_1$  and  $\alpha_2$ , as follows. Processes in  $Q$  take no steps in  $\alpha_3$ . The original names of processes in  $P$  are  $I_1$ , and original names of processes in  $P'$  are  $I_2$ . The execution is identical to  $\alpha_1$  from time 0 until time  $t(I_1)$ , and to  $\alpha_2$  from this time until time  $t(I_1) + t(I_2)$ . All messages sent from processes in  $P$  to processes in  $P'$  and from processes in  $P'$  to processes in  $P$  are delivered after time  $t(I_1) + t(I_2)$ .

In  $\alpha_3$ , processes in  $P$  do not receive messages from processes in  $P' \cup Q$ . Furthermore,  $P' \not\rightarrow P$ ; i.e., processes in  $P'$  cannot read what processes in  $P$  wrote to the shared memory. Hence,  $\alpha_3$  is indistinguishable to  $p_{i_j}$  from  $\alpha_1$ , and hence, it picks new name  $r$ . Similarly,  $\alpha_3$  is indistinguishable to  $p'_{i_j}$  from  $\alpha_2$ , and hence, it also picks new name  $r$ , which contradicts the uniqueness of new names.  $\blacktriangleleft$

## 5 The M&M Model

In the m&m model [5, 23], the shared memory connections are defined by a *shared-memory domain*  $L$ , which is a collection of sets of processes. For each set  $S \in L$ , all the processes in the set may share any number of registers among them. Our model when the shared memory has no access restrictions is a dual of the general m&m model, and they both capture the same systems. We say that  $L$  is *uniform* if it is induced by an undirected *shared-memory graph*  $G = (V, E)$ , where each vertex in  $V$  represents a process  $p$ . For every process  $p$ ,  $S_p = \{p\} \cup \{q : (p, q) \in E\}$ , then  $L = \{S_p : p \text{ is a process}\}$ . In the uniform m&m model each memory is associated with a process  $p$ , and all the processes in  $S_p$  may access it. That is, a process can access its own memory and the memories of its neighbors.

In the m&m model, there are no access restrictions on the shared memory. Hence, for every process  $p$ ,  $|R_p| = |W_p| = |S_p|$ . Therefore,  $\rho = \sum_{\text{process } p} |S_p| = \sum_{\text{process } p} d(p) + 1 = 2|E| + n = O(n^2)$  and  $\sigma = O(n^3)$ , where  $d(p)$  is the degree of process  $p$  in the graph. Substituting into the algorithms presented in Section 4, we obtain polynomial complexity for all of them, including a polynomial randomized consensus algorithm. In the general m&m model,  $\rho$  and  $\sigma$  are unbounded.

► **Definition 12** ([23]). *Given a shared-memory domain  $L$ ,  $f_L$  is the largest integer  $f$  such that for all process subsets  $P$  and  $P'$  of size  $n - f$  each, either  $P \cap P' \neq \emptyset$  or there is a set  $S \in L$  that contains both a process from  $P$  and a process from  $P'$ .*

Hadzilacos, Hu and Toueg [23] show that an SWMR register can be implemented in the m&m model if and only if at most  $f_L$  process may fail. Therefore in the m&m model,  $f_{opt} = f_L$ . We can see the connection between the two definitions by observing that, in this model,  $p \leftrightarrow q$  if  $p = q$  or there is a set  $S \in L$  such that  $p, q \in S$ . We simply write  $\leftrightarrow$ , since the shared memory has no access restrictions.

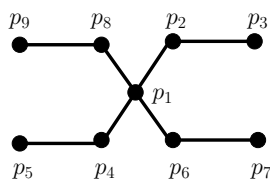
The *square* of a graph  $G = (V, E)$  is the graph  $G^2 = (V, E^2)$ , where  $E^2 = E \cup \{(u, v) : \exists w \in E \text{ such that } (u, w) \in E \text{ and } (w, v) \in E\}$ . I.e., there is an edge in  $G^2$  between every two vertices that are in distance at most 2 in the graph  $G$ .

► **Definition 13** ([23]). *Given an undirected graph  $G = (V, E)$ ,  $f_G$  is the largest integer  $f$  such that for all subsets  $P$  and  $P'$  of  $V$  of size  $n - f$  each, either  $P \cap P' \neq \emptyset$  or  $G^2$  has an edge  $(u, v)$  such that  $u \in P$  and  $v \in P'$ .*

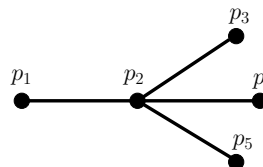
In the uniform m&m model,  $f_L = f_G = f_{opt}$  [23], and  $p \leftrightarrow q$  if  $p = q$  or  $(p, q)$  is an edge in  $G^2$ .

We have seen that  $f_{opt} \leq f_{maj}$ . Figure 1 shows a graph where  $f_{opt} < f_{maj}$ . Thus, the converse inequality does not hold in the (uniform or general) m&m model.

► **Definition 14** ([5]). *A process  $p$  represents itself and all its neighbors, that is,  $\{p\} \cup \{q : (p, q) \in E\}$ . A set of processes  $P$  represents the union of all the processes represented by processes in  $P$ .*



■ **Figure 1** Counter example.



■ **Figure 2** Counter example for  $n = 5$ .

Aguilera et al. [5] present a randomized consensus algorithm, called HBO, which is based on Ben-Or's algorithm [15]. Like Ben-Or's algorithm, HBO has exponential time and message complexities. HBO assumes that the nonfaulty processes represent a majority of the processes. Below, we show that the resilience of the HBO algorithm is not optimal. We first capture the condition required for the correctness of the HBO algorithm, with the next definition.

► **Definition 15.**  $f_{m\&m}$  is the largest integer  $f$  such that every set  $P$  of  $n - f$  processes represents a majority of the processes.

It can be shown that  $f_{m\&m} \leq f_{opt}$ . On the other hand, for every  $n > 4$ , there is a shared-memory graph, such that  $f_{m\&m} < f_{opt}$  in the uniform m&m model. The graph is the star graph over  $n$  vertices, and has edges  $\{(p_1, p_2)\} \cup \{(p_2, p_i) : 3 \leq i \leq n\}$ . (See Figure 2, for  $n = 5$ .) Thus, requiring at least  $n - f_{m\&m}$  nonfaulty processes is strictly stronger than requiring  $n - f_{opt}$  nonfaulty processes. Therefore, the HBO algorithm does not have optimal resilience. Intuitively this happens since HBO does not utilize all the shared-memory connections that are embodied in  $G^2$ . Thus, our algorithm (Section 4.3), has better resilience than HBO, which we show is optimal, in addition to having polynomial complexity.

Aguilera et al. [5] also present a lower bound on the number of failures any consensus algorithm can tolerate in the m&m model. To state their bound, consider a graph  $G = (V, E)$ , and let  $B, S$  and  $T$  be a partition of  $V$ .  $(B, S, T)$  is an *SM-cut* in  $G$  if  $B$  can be partitioned into two disjoint sets  $B_1$  and  $B_2$ , such that for every  $b_1 \in B_1, b_2 \in B_2, s \in S$  and  $t \in T$ , we have that  $(s, t), (b_1, t), (b_2, s) \notin E$ .

► **Theorem 16** ([5]). *Consensus cannot be solved in the uniform m&m model in the presence of  $f$  failures if there is a SM-cut  $(B, S, T)$  such that  $|S| \geq n - f$  and  $|T| \geq n - f$ .*

Although the resilience of HBO is not optimal, we show that this lower bound on resilience is optimal, by proving that if a system is  $f$ -partitionable then the condition in Theorem 16 holds. By Theorem 9, these two conditions are equal in the m&m model.

► **Theorem 17.** *In the uniform m&m model, if the system is  $f$ -partitionable then there is an SM-cut  $(B, S, T)$  with  $|S| \geq n - f$  and  $|T| \geq n - f$ .*

## 6 The Cluster-based Model

In the hybrid, *cluster-based* model of [28, 31], processes are partitioned into  $m, 1 \leq m \leq n$ , non-empty and disjoint subsets  $P_1, \dots, P_m$ , called *clusters*. Each cluster has an associated shared memory; only processes of this cluster can (atomically) read from and write to this shared memory. The set of processes in the cluster of  $p$  is denoted  $cluster[p]$ . As in the m&m model, there are no access restrictions on the shared memory. Hence,  $|R_p| = |W_p| = 1$  for every process  $p$ , and therefore,  $\rho = n$  and  $\sigma = O(n^2)$ .

In the cluster-based model,  $p \leftrightarrow q$  if and only if  $p$  and  $q$  are in the same cluster.

If  $p \leftrightarrow q$  and  $q \leftrightarrow w$ , for some processes  $p, q$  and  $w$ , then  $p$  and  $q$  are in the same cluster and  $q$  and  $w$  are in the same cluster. Since clusters are disjoint, it follows that  $p$  and  $w$  are in the same cluster, implying that  $\leftrightarrow$  is transitive.

► **Definition 18.**  $f_{cluster}$  is the largest integer  $f$  such that for all sets of processes  $P$  and  $P'$ , each of size  $(n - f)$ , either  $P \cap P' \neq \emptyset$  or some cluster contains a process in  $P$  and a process in  $P'$ .

► **Observation 1.** In the cluster-based model  $f_{opt} = f_{cluster}$ .

► **Lemma 19.** In the cluster-based model,  $f_{opt} = f_{maj}$ .

► **Lemma 20.** In the cluster-based model, for every two sets of processes,  $P$  and  $Q$ , and  $f \leq f_{opt}$ , if  $|\vec{P}| \geq n - f$  and  $|\vec{Q}| \geq n - f$  then  $P \leftrightarrow Q$ .

Raynal and Cao [31] present two randomized consensus algorithms for the cluster-based model. One is also based on Ben Or's algorithm [15], using local coins, and the other is based on an external common coin (whose implementation is left unspecified). These algorithms terminate in an execution if there are distinct clusters whose total size is (strictly) larger than  $n/2$ , each containing at least one nonfaulty process. Clearly, if  $f \leq f_{maj}$ , this condition holds for every execution with at most  $f$  failures. Since  $f_{opt} \leq f_{maj}$ , the condition holds if there are at most  $f \leq f_{opt}$  failures. Lemma 19 implies that these two definitions are equivalent by proving that  $f_{opt} = f_{maj}$ . This means that the maximum resilience guaranteeing that every two sets of nonfaulty processes can communicate is equal to the one guaranteeing that every set of nonfaulty processes can communicate with a majority of the processes.

In the cluster-based model, if a process  $p \in P_i$  does not crash then all other processes receive the information from all the processes of  $P_i$ , as if none of them crashed. For this reason, we say that  $p$  represents all processes in  $P_i$  (note that this definition is different than Definition 14). If a process  $q$  receives messages from processes representing  $k$  clusters  $P_1, \dots, P_k$ , such that  $|P_1| + \dots + |P_k| > n/2$ , then it has received information from a majority of the processes. This observation does not change the resilience threshold, i.e., the maximal number of failures that can be tolerated, but allows to wait for a smaller number of messages, thereby, making the algorithm execute faster. Lemma 20 proves that every two sets of processes representing at least  $n - f_{opt}$  processes can communicate. Therefore, instead of waiting for a majority of represented processes, as is done in [31], it suffices to wait for  $n - f_{opt}$  represented processes. Since  $n - f_{opt} \leq \lfloor n/2 \rfloor + 1$ , this means that in some cases it suffices to wait for fewer than a majority of represented processes.

This is not the case in the m&m model. For example, in the graph of Figure 1,  $f_{opt} = 6$ . For  $P = \{p_7, p_9\}$ ,  $\vec{P} = \{p_1, p_6, p_7, p_8, p_9\}$ , and for  $Q = \{p_3, p_5\}$ ,  $\vec{Q} = \{p_1, p_2, p_3, p_4, p_5\}$ , so  $|\vec{P}| = |\vec{Q}| = 5 > n/2$ , but  $P \not\leftrightarrow Q$ . Therefore, even though the system is not  $f$ -partitionable, and the set of non-faulty processes can communicate with a majority of the processes, it does not suffice to wait for more than  $n/2$  represented processes.

## 7 Discussion

This paper studies the optimal resilience for various problems in mixed models. Our approach builds on simulating a SWMR register, which allows to investigate the resilience of many problems, like implementing MWMR registers and atomic snapshots, or solving randomized consensus, approximate agreement and renaming. Prior consensus algorithms for mixed models [5, 31] start from a pure message-passing algorithm and then try to exploit the



added power of shared memory. In contrast, we start with a shared-memory consensus algorithm and systematically simulate it in the mixed model. This simplifies the algorithms and improves their complexity, while still achieving optimal resilience.

It would be interesting to investigate additional tasks and objects. An interesting example is *k-set consensus* [19], in which processes must decide on at most  $k$  different values. This is trivial for  $k = n$  and reduces to consensus, for  $k = 1$ . For the pure message-passing model, there is a  $k$ -set consensus algorithm [19], when the number of failures  $f < k$ . This bound is necessary for solving the problem in shared memory systems [17, 26, 32]. Since resilience in a mixed system cannot be better than in the shared-memory model, it follows that  $f < k$  is necessary and sufficient for any mixed model. Thus, when  $f_{opt} < k - 1$ , a system can be  $f$ -partitionable and still offer  $f$ -resilience for  $k$ -set consensus.<sup>1</sup>

The weakest failure detector needed for implementing a register in the cluster-based model is strictly weaker than the weakest failure detector needed in the pure message-passing model [28]. This aligns with the improved resilience we can achieve in a mixed model compared to the pure message-passing model. It is interesting to explore the precise improvement in resilience achieved with specific failure detectors and other mixed models.

We would also like to study systems where the message-passing network is not a clique.

---

## References

- 1 Gen-Z draft core specification. <https://genzconsortium.org/specification/gen-z-core-specification-1-1-draft/>. Accessed: 2020-08-26.
- 2 InfiniBand. <https://www.infinibandta.org/about-infiniband/>. Accessed: 2020-08-26.
- 3 iWARP. <https://en.wikipedia.org/wiki/IWARP>. Accessed: 2020-08-26.
- 4 RDMA over converged ethernet. [https://en.wikipedia.org/wiki/RDMA\\_over\\_Converged\\_Ethernet](https://en.wikipedia.org/wiki/RDMA_over_Converged_Ethernet). Accessed: 2020-08-26.
- 5 Marcos K. Aguilera, Naama Ben-David, Irina Calciu, Rachid Guerraoui, Erez Petrank, and Sam Toueg. Passing messages while sharing memory. In *PODC*, page 51–60, 2018.
- 6 James Aspnes and Maurice Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–461, September 1990.
- 7 Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, January 1995.
- 8 Hagit Attiya, Amotz Bar-Noy, Danny Dolev, David Peleg, and Rüdiger Reischuk. Renaming in an asynchronous environment. *Journal of the ACM*, 37(3):524–548, 1990.
- 9 Hagit Attiya and Constantin Enea. Putting strong linearizability in context: Preserving hyperproperties in programsthat use concurrent objects. In *DISC*, pages 2:1–2:17, 2019.
- 10 Hagit Attiya and Arie Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM J. Comput.*, 31(2):642–664, 2001. doi:10.1137/S0097539700366000.
- 11 Hagit Attiya, Arie Fouren, and Eli Gafni. An adaptive collect algorithm with applications. *Distributed Computing*, 15(2):87–96, 2002.
- 12 Hagit Attiya, Nancy A. Lynch, and Nir Shavit. Are wait-free algorithms fast? *J. ACM*, 41(4):725–763, 1994. doi:10.1145/179812.179902.
- 13 Hagit Attiya and Ophir Rachman. Atomic snapshots in  $O(n \log n)$  operations. *SIAM J. Comput.*, 27(2):319–340, 1998.

---

<sup>1</sup> There is a lower bound of  $k > \frac{n-1}{n-f}$  for pure message-passing systems, proved using a partitioning argument [16]. It might seem that adding shared memory will allow to reduce this bound, however, this is not the case, since for the relevant ranges of  $k$  ( $1 < k < n$ ), the bound on the number of failures implied from this bound is at least  $k$ .

- 14 Amotz Bar-Noy and Danny Dolev. A partial equivalence between shared-memory and message-passing in an asynchronous fail-stop distributed environment. *Math. Syst. Theory*, 26(1):21–39, 1993. doi:10.1007/BF01187073.
- 15 Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *PODC*, pages 27–30, 1983.
- 16 Martin Biely, Peter Robinson, and Ulrich Schmid. Easy impossibility proofs for  $k$ -set agreement in message passing systems. In *OPODIS*, pages 299–312, 2011.
- 17 Elizabeth Borowsky and Eli Gafni. Generalized FLP impossibility result for  $t$ -resilient asynchronous computations. In *STOC*, pages 91–100, 1993.
- 18 Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824–840, 1985.
- 19 Soma Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Inf. Comput.*, 105(1):132–158, 1993. doi:10.1006/inco.1993.1043.
- 20 Faith E. Fich, Maurice Herlihy, and Nir Shavit. On the space complexity of randomized synchronization. *Journal of the ACM*, 45(5):843–862, 1998. doi:10.1145/290179.290183.
- 21 Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985. doi:10.1145/3149.214121.
- 22 Wojciech Golab, Lisa Higham, and Philipp Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In *STOC*, page 373–382, 2011.
- 23 Vassos Hadzilacos, Xing Hu, and Sam Toueg. Optimal register construction in m&m systems. In *OPODIS*, pages 28:1–28:16, 2019.
- 24 Vassos Hadzilacos, Xing Hu, and Sam Toueg. Optimal register construction in m&m systems (version 3). *CoRR*, abs/1906.00298, 2020. URL: <http://arxiv.org/abs/1906.00298>.
- 25 Vassos Hadzilacos, Xing Hu, and Sam Toueg. Randomized consensus with regular registers. *CoRR*, abs/2006.06771, 2020. URL: <http://arxiv.org/abs/2006.06771>.
- 26 Maurice Herlihy and Nir Shavit. The asynchronous computability theorem for  $t$ -resilient tasks. In *STOC*, pages 111–120, 1993.
- 27 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.
- 28 Damien Imbs and Michel Raynal. The weakest failure detector to implement a register in asynchronous systems with hybrid communication. *Theor. Comput. Sci.*, 512:130–142, 2013. doi:10.1016/j.tcs.2012.06.030.
- 29 Leslie Lamport. On interprocess communication—part I: Basic formalism. *Distributed Computing*, pages 77–85, 1986.
- 30 Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. *SIGARCH Comput. Archit. News*, 37(3):267–278, June 2009. doi:10.1145/1555815.1555789.
- 31 Michel Raynal and Jiannong Cao. One for all and all for one: Scalable consensus in a hybrid communication model. In *ICDCS*, pages 464–471, 2019.
- 32 Michael E. Saks and Fotios Zaharoglou. Wait-free  $k$ -set agreement is impossible: the topology of public knowledge. In *STOC*, pages 101–110, 1993.
- 33 Paul M. B. Vitányi and Baruch Awerbuch. Atomic shared register access by asynchronous hardware. In *FOCS*, pages 233–243, 1986.

# CSR++: A Fast, Scalable, Update-Friendly Graph Data Structure

## Soukaina Firmlı

Mohammed V University in Rabat, Ecole Mohammadia d'Ingénieurs, SIP Research Team, Morocco  
Oracle Labs, Casablanca, Morocco  
soukaina.firmlı@oracle.com

## Jean-Pierre Lozi

Oracle Labs, Zürich, Switzerland  
jean-pierre.lozi@oracle.com

## Alexander Weld

Oracle Labs, Zürich, Switzerland  
alexander.weld@oracle.com

## Sungpack Hong

Oracle Labs, Palo Alto, CA, USA  
sungpack.hong@oracle.com

## Vasileios Trigonakis

Oracle Labs, Zürich, Switzerland  
vasileios.trigonakis@oracle.com

## Iraklis Psaroudakis

Oracle Labs, Zürich, Switzerland  
iraklis.psaroudakis@oracle.com

## Dalila Chiadmi

Mohammed V University in Rabat, Ecole Mohammadia d'Ingénieurs, SIP Research Team, Morocco  
chiadmi@emi.ac.ma

## Hassan Chafi

Oracle Labs, Palo Alto, CA, USA  
hassan.chafi@oracle.com

---

## Abstract

The graph model enables a broad range of analysis, thus graph processing is an invaluable tool in data analytics. At the heart of every graph-processing system lies a concurrent graph data structure storing the graph. Such a data structure needs to be highly efficient for both graph algorithms and queries. Due to the continuous evolution, the sparsity, and the scale-free nature of real-world graphs, graph-processing systems face the challenge of providing an appropriate graph data structure that enables both fast analytical workloads and low-memory graph mutations. Existing graph structures offer a hard trade-off between read-only performance, update friendliness, and memory consumption upon updates. In this paper, we introduce CSR++, a new graph data structure that removes these trade-offs and enables both fast read-only analytics and quick and memory-friendly mutations. CSR++ combines ideas from CSR, the fastest read-only data structure, and adjacency lists to achieve the best of both worlds. We compare CSR++ to CSR, adjacency lists from the Boost Graph Library, and LLAMA, a state-of-the-art update-friendly graph structure. In our evaluation, which is based on popular graph-processing algorithms executed over real-world graphs, we show that CSR++ remains close to CSR in read-only concurrent performance (within 10% on average), while significantly outperforming CSR (by an order of magnitude) and LLAMA (by almost 2×) with frequent updates.

**2012 ACM Subject Classification** Information systems → Data structures; Theory of computation → Concurrency; Theory of computation → Graph algorithms analysis; Computing methodologies → Concurrent algorithms

**Keywords and phrases** Data Structures, Concurrency, Graph Processing, Graph Mutations

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2020.17

## 1 Introduction

Graph processing is an invaluable tool for data analytics, as illustrated by the plethora of relatively recent work aiming at achieving high performance for graph algorithms [12, 17, 23, 34, 35, 41], such as PageRank [29], or graph querying/mining [13, 21, 26, 27, 31, 33, 38], e.g., using PGQL [5]. At the heart of each graph system lies the graph data structure, responsible for holding the vertices and the edges comprising the graph, and whose performance largely



© Soukaina Firmlı, Vasileios Trigonakis, Jean-Pierre Lozi, Iraklis Psaroudakis, Alexander Weld, Dalila Chiadmi, Sungpack Hong, and Hassan Chafi;  
licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 17; pp. 17:1–17:16



Leibniz International Proceedings in Informatics

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

contributes to the general performance of the system. The ideal graph structure should offer excellent read-only performance, fast mutations (i.e., vertex or edge insertions and deletions), and low memory consumption with or without mutations.

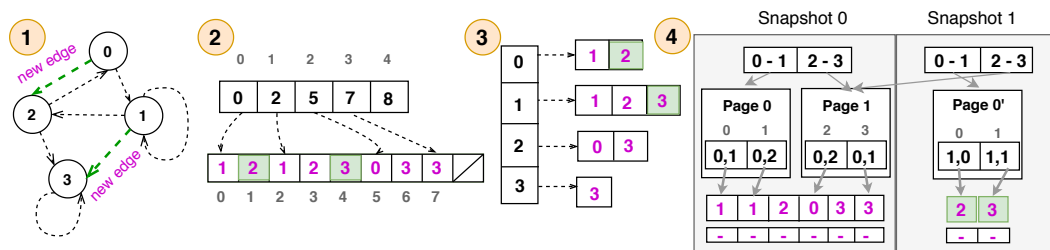
Classic graph data structures typically trade some characteristics for others (see Section 2.1). Adjacency lists enable quick graph updates and consume relatively little memory, but sacrifice performance, as they lead to expensive pointer chasing. Adjacency matrices enable quick edge updates, but sacrifice vertex insertions and consume a lot of memory. Finally, the Compressed Sparse Row (CSR) representation offers a good memory footprint with excellent read-only performance by completely sacrificing mutability: even a single vertex or edge insertion requires complete reallocation of the underlying structures. There have been efforts to improve the update-friendliness of CSR (see Section 2.2). These include in-place update techniques [36, 39], batching techniques [11, 24], and changeset-based updates with delta maps [22] and multi-versioning [23]. A multi-versioning solution is used in LLAMA [23], a state-of-the-art update-friendly graph structure that enables mutability on top of CSR by appending delta snapshots. However, having a frequent flow of graph updates – which is the common case in real-life scenarios, such as financial transactions – results in a large number of delta logs, and thus high memory utilization and decreased performance. Compaction operations on these data structures are often expensive, hindering the benefits of fast mutability (see Section 4 for a performance analysis). Very often, users simply need to operate on the most up-to-date version of the graph data, thus asking for fast in-place graph updates.

In this paper, we introduce CSR++, a concurrent graph data structure with performance comparable to CSR, efficient updates, and memory consumption proportional to the number of mutations. CSR++ maintains the array-continuity that makes CSR very fast. In particular, vertices are stored in arrays, segmented for better update-friendliness. As in CSR, vertex IDs in CSR++ are implicitly determined by the location of the vertex, but include both the segment ID and where in the segment the vertex lies. Accordingly, the 64 bits of vertex IDs are split into `{int segment_id; int in_segment_id}`, making vertices directly addressable. Due to segmentation, inserting a new vertex is as simple as (i) if needed, appending a new segment to the array of segments, and (ii) appending the vertex to that segment.

In contrast to CSR, and like adjacency lists, CSR++ can independently manage the edges of each vertex. If a vertex has two or more edges, CSR++ holds a pointer to an array storing the edges. To reduce memory usage, for single-edge vertices, the target vertex of the edge is inlined in lieu of the array pointer. All in all, CSR++ maintains the array-oriented structures of CSR for performance, while enabling per-vertex edge-list modifications to enable fast updates as with adjacency lists.

Apart from vertices and edges, graph structures also need to store vertex and edge properties, which are a prominent feature of property graphs. CSR++ includes segmentation techniques to enable fast property updates when new vertices or edges are inserted. Vertex properties are stored in segmented arrays, and each vertex holds a pointer to an array of edge property values, allowing for fast per-segment or per-vertex reallocation of property arrays.

We evaluate CSR++ with both read and update workloads, with various graphs and graph algorithms, and compare it against CSR, adjacency lists, and LLAMA. Our results indicate that CSR++ is much faster than adjacency lists, is almost as fast as CSR on read-only workloads, and has faster updates and lower memory consumption than LLAMA. In particular, CSR++ performs on average within 10% of the read-only performance of CSR with 36 threads and is an order of magnitude faster for updates. Furthermore, CSR++ is faster than LLAMA for most read-only workloads, is almost  $2\times$  faster in applying batched updates, and consumes  $4\times$  less memory when 100 update batches are applied on a base graph.



**Figure 1** (1) An example graph with newly inserted edges in green, represented in different graph structures: (2) CSR, (3) Adjacency list, and (4) LLAMA with implicit linking and deletion vectors.

The main contributions of this paper are as follows:

- CSR++, a new graph data structure that supports fast in-place updates, without sacrificing read-only performance or memory consumption; and
- Our thorough evaluation that shows that CSR++ achieves the best of both read-only and update-friendly worlds.

## 2 Background & Related Work

Graphs are already a prominent data model, especially in the current era of big data and data deluge [16]. The advantage over the traditional relational model is that graphs can inherently model entities and their relationships. While a relational model needs to join tabular data in order to process foreign-key relationships, graph-processing engines have built-in ways to efficiently iterate over the graph [37], e.g., over the neighbors of vertices, and support a plethora of expressive graph algorithms (such as Green-Marl [19, 34]) and graph pattern-matching queries (such as PGQL [5], SPARQL [8], and Gremlin [9]).

Graphs can be represented with different models and data representations. A popular model is the RDF (Resource Description Framework) graph data model [8], which became popular with the rise of the semantic web [10]. RDF regularizes the graph representation as a set of triples. RDF adds links for all data, including constant literals, and it does not explicitly store vertices, edges, or properties separately. As the graph is not stored in its native format, it results in reduced performance [40], as RDF engines are forced to process and join a large number of intermediate results.

Our paper focuses on a more recent model, the Property Graph (PG) model [6, 38], which is widely adopted by various graph databases and processing systems (such as Neo4J [27] and PGX [28, 31]). PG represents the topology of a graph natively as vertices and edges, and stores properties separately in the form of key-value pairs. This separation allows for quick traversals over the graph structure. Classic graph algorithms, such as PageRank [29] and Connected Components, are very naturally expressed on top of property graphs [34].

In order for graph-processing engines to provide efficient solutions for large-scale graphs, they rely on efficient data structures, potentially resident on main memory [23, 41, 18, 15], to store and process vertices and their relationships. One of the key challenges for in-memory graph-processing engines is to design data structures with reasonable memory footprint [23] that can support fast graph algorithm execution [19] and query pattern matching [32], whilst supporting topological modifications (like additions or removals of vertices and edges), either in batches or in a streaming fashion [25, 11]. In the following, we discuss the most prominent data structures in related work [15], and motivate the necessity of the novel CSR++. We show in Figure 1 an example of a graph and how it is represented in different formats.

## 2.1 Graph Representations

**Adjacency Matrices and Lists.** An adjacency matrix represents a graph with a  $V^2$  matrix  $M$ , where  $V$  is the number of vertices in the graph. A non-zero cell  $M[v_s][v_d]$  represents the directed edge from a source vertex  $v_s$  to a destination vertex  $v_d$ . An adjacency matrix is not preferred for sparse graphs, i.e., graphs where the number of edges  $E \ll V^2$ , due to increased memory footprint and decreased performance in analytics.

Adjacency lists represent the graph with a set of vertices, where each vertex is associated with a list of neighbors, as shown in Figure 1(3). An adjacency list typically consumes less memory than an adjacency matrix, since for a given vertex only the existing edges need to be stored. The typical format for the adjacency list uses linked lists, with extra pointers, but more cache-friendly variants exist, such as Blocked Adjacency Lists, where adjacencies are represented by simple arrays [41] or with linked lists of buckets containing a fixed size of edges [14, 17]. As an example, the popular Boost C++ Library [1] implements adjacency lists and the edge structures can be configured to either be vectors, lists, or sets. Although adjacency lists can be efficient in terms of mutations, they struggle in read-only workloads, as we show in Section 4.

**Compressed Sparse Row (CSR).** CSR [19] is a commonly used data structure for sparse graphs, because it compacts adjacencies into two arrays: The vertex array and the edge array. In the vertex array, each vertex is identified by its array index. The vertex cell stores the begin offset in the edge array (the end offset is implicit, as it is equal to the begin offset of the next vertex cell), where the list of the destination neighbors of this vertex is stored, as shown in Figure 1(2). In terms of graph mutations, CSR is very inefficient. For example, to add an edge, the whole edge array needs to be reallocated with the newly-added edge and the subsequent edges shifted by one place.

## 2.2 Graph Mutations

Graph mutations, or updates, mostly refer to vertex or edge insertions and deletions. Although CSR is one of the most popular data structures for representing a graph, it is, as mentioned above, very limiting for graph mutations. This has prompted a lot of related work on mutable data structures to represent graphs that can efficiently digest sets of updates.

**In-Place Updates.** Techniques that use in-place updates employ the aforementioned static data structures in a way that allows for in-place digestion of sets with insertions and deletions of vertices and edges, without requiring the expensive rebuild of the data structure. For instance, Dense [20] is a concurrent graph adjacency matrix that supports mutations and partial traversals through a coordination protocol, but does not handle graph properties. NetworKit [36], in order to perform edge insertions, stores adjacencies vectors that double the size of the initial array to reserve enough space for new incoming edges. Madduri et al. [24] use the same underlying technique but define a configurable size of the new edge array instead of using factor 2. Ediger et al. [14] implement blocked adjacency lists and allow insertions by appending new blocks and updating pointers. Wheatman et al. [39] implement a variant of CSR that leaves space at the end of each adjacency list to allow efficient single-threaded mutations. We employ similar techniques in CSR++ to ingest mutations, but in a parallel manner while also handling graph property mutations.

**Batching.** Regarding the sources of changes, they can be continuous streams of updates [11, 14] or single changes applied as “local” mutations. Generally, when applying a batch of updates, frameworks perform pre-processing to re-arrange the batches in ways that can speed-up the mutations. For instance, Madduri et al. [24] apply techniques on the list of new edges, such as sorting, re-ordering, and partitioning, in order to exploit parallelism at the time of the changes application. Similarly, CSR++ groups updates by their source vertices, and uses multiple threads to perform fast edge insertions (see Section 3.2).

**Multi-Versioning & Deltas.** One way to extend CSR to support fast updates is by allocating a separate structure to store only the new changes [22] in delta maps. Furthermore, by using deltas, the following systems can run analytical workloads on different static versions (snapshots) of the changing graph over time. LLAMA [23] is a state-of-the-art snapshot-based graph system that supports multi-versioning by storing deltas as separate snapshots and supports concurrent access to those snapshots (see Figure 1(4)). ASGraph [17] limits its read-access to one snapshot at a time but still ensures high performance by extending its underlying data structure [14] with temporal attributes. Graphite [30] is an in-memory relational column-store that employs also multi-versioning snapshots using deltas.

The downside of the above approaches is two-fold. First, maintaining separate snapshots increases the memory requirements of the system, as a frequent flow of graph updates results in a large number of deltas. Second, the performance of analytics is degraded because they need to read from both the original structure and the deltas and reconcile them. A solution to the potential performance degradation is to periodically merge the delta maps into CSR, an operation called compaction. Compaction, however, can become very expensive, often zeroing the mutability performance benefits of these structures. For users that wish to operate on the most up-to-date version of the graph data, we show that CSR++, which is designed for in-place graph mutations, achieves better analytics and update performance than LLAMA [23], with up to an order of magnitude lower memory requirements (see Section 4).

### 3 CSR++: Design and Implementation

With CSR++, our goal is to design a data structure that stores graphs and allows fast in-place mutations with analytics performance comparable to CSR. In order to allow for fast algorithms, CSR++ enables fast concurrent accesses to the main graph data (vertex and edge tables) and stores additional graph data, such as reverse edges, user-defined keys, and vertex and edge properties. CSR++ does not aim to support versioning, but instead fast in-place updates, allowing to withstand frequent small updates without the overhead of snapshots.

#### 3.1 Graph Topology and Properties

CSR++ is a concurrent structure that stores the graph in memory using segmentation techniques. It allows in-place insertions by allocating additional space for new incoming edges and supports logical deletions of vertices and edges. Figure 2 shows the building blocks of CSR++.

**Segments.** CSR++ stores vertices in arrays called *segments*. The graph is represented as an array of segments, each storing a fixed number of vertices defined by a global configurable parameter `NUM_V_SEG`. Segments give flexibility to CSR++ in three ways: (i) memory allocations and reallocations use segment granularity, (ii) vertex properties are allocated per segment, and (iii) synchronization for concurrency uses segment granularity. As with CSR, CSR++ packs the vertices in arrays to reduce the memory footprint when storing sparse graphs,



which also results in better cache locality. The entry point to CSR++ is an array that stores all segments; this also enables quick segment additions. Finally, each segment stores a vector of pointers to the vertex property arrays.

**Vertices.** Each vertex stores its degree, a pointer to its list of neighbors, and optionally a pointer to the property values of its edges. This design resembles a mix of CSR and adjacency lists, however, adding a new vertex in CSR++ is faster (see Section 3.2) considering that the vertex array is segmented, i.e., we do not need to copy the whole vertex array to add or remove entries. CSR++ does not store explicit IDs for vertices nor edges, but since all segments store a fixed number `NUM_V_SEG` of vertices, we can compute implicit IDs for vertices using the segment ID and the index of the vertex in the segment: `global_v_id = (seg_id * NUM_V_SEG) + v_id`. Overall, the vertex structure consists of the following fields:

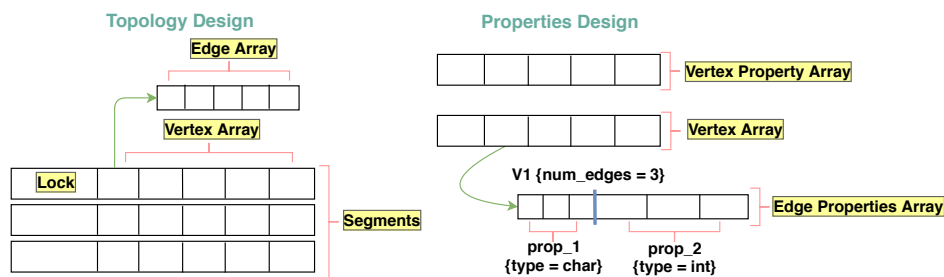
- `length` (4 bytes): The vertex degree. A length of `-1` indicates a deleted vertex.
- `neighbors` (8 bytes): A pointer to the set of neighbors. As a space optimization, if `length = 1`, this field directly contains the neighbor’s vertex ID.
- `edge_properties` (8 bytes): A pointer to the set of edge properties. As a space optimization, this field can be disabled in case the graph does not define edge properties.

**Edges.** CSR++ represents the neighbors list of a vertex by an array of edges, where every entry stores the coordinates (i.e., the vertex ID and the segment ID) of the corresponding neighbor. At loading time, the edges are sorted; as with CSR and LLAMA, keeping the edges sorted allows for better cache performance. Moreover, this semi-sorting is necessary for CSR++ in a deletion-frequent context, as we use binary search to locate edges. Additionally, as an optimization for update-friendliness, CSR++ can be configured to create extra empty space for new incoming edges during graph loading (see Section 3.2). The edge structure consists of the following fields:

- `deleted_flag` (2 bytes): For logical deletion of edges.
- `vertex_id` (2 bytes): The index of the neighbor in the segment; using 16 bits allows for segments with a capacity `NUM_V_SEG` of up to 65536 entries.
- `segment_id` (4 bytes): The segment ID where the neighbor is stored.

For better cache utilization when scanning over vertices and better load balancing when using multiple threads, the number of vertices that a segment stores should neither be very small nor too large, in order to avoid copying large amounts of data when the graph is updated. By default we use `NUM_V_SEG = 4096` vertices per segment.

**Properties.** Vertex property values are stored in arrays parallel to the vertices array. CSR++ keeps a vector of pointers to each vertex property array within the segment. The size of each array is therefore `NUM_V_SEG * sizeof(Property_Type)`. For edge properties, we use the



■ **Figure 2** The building blocks of CSR++: Graph topology (left) and graph properties (right).

same segmentation approach as vertices. If the user enables edge properties, each vertex structure stores a pointer to an array of edge property values, as shown in Figure 2. In case of multiple properties, we allocate an array that stores the values for different edge properties in a cache-aligned manner. In order to locate a specific edge property  $p$ , we use offsets and the position of its values can be calculated given the type of that property  $T_p$ , the index  $i$  of the edge in the neighbor list, and the degree  $d$  of the vertex  $v$ . For example, suppose the user registers  $n$  edge properties, then the total size of the edge properties of a vertex  $v$  is  $\sum_{p=1}^n (\text{sizeof}(Type_p) * d)$ . Similarly, the values of the  $x$ th property begin at  $Values(x) = \sum_{p=1}^x (\text{sizeof}(Type_p) * d)$ . Accordingly, the property value for the  $x$ th property of the edge  $i$  is  $Value(x, i) = Values(x) + (i * \text{sizeof}(Type_x))$ .

The reason for this choice is that having the edge properties stored in parallel to the edge arrays allows to copy-on-write the edge property arrays of the updated vertices only, unlike with CSR where there is a need to rebuild edge properties for the entire graph. In addition, this design makes it easier to keep the property values in the same order as the edges in case we have to sort them after an update operation. As we show in Section 4.6, this design adds a moderate memory overhead. Naturally, if the to-be-loaded graph configuration does not include edge properties, edge property support can be disabled to save memory.

**Additional Structures.** Most real-life graphs include user-provided vertex IDs, e.g., a full-name string. CSR++ supports mapping of user vertex keys to internal IDs by storing them in a map and, inversely, internal IDs are mapped directly inside the segments of CSR++ using one ID mapping array per segment. For directed graphs, some algorithms, e.g., PageRank, require access to reverse edges and sometimes mappings from reverse to their corresponding forward edges (e.g., Weighted PageRank; see Section 4). To ensure fast lookup over the reverse edges and their mapping, similar to most representations, such as CSR in Green-Marl [2] and LLAMA, CSR++ reserves additional structures to store the reverse edges corresponding to each forward edge, as well as the mapping between their indices stored as an edge property. These increase the memory footprint but contribute to higher performance.

**Synchronization.** Synchronization in CSR++ is implemented at the segment level, using spinlocks to protect data writes. CSR++ does not support scans concurrent to updates.

### 3.2 Update Protocols

CSR++ supports efficient concurrent in-place mutations by allowing both single local updates (e.g., inserting edge by edge) and batch update operations.

**Vertex and Edge Insertion.** For vertex insertions, as described in the previous section, the `length` field in the vertex structure stores the degree of the vertex. Lengths  $\geq 0$  indicate a valid vertex. New vertex insertions land in the last segment. To add a vertex: in case there is enough space in the last segment, CSR++ finds the first non-valid vertex, and then sets the vertex accordingly. Setting the vertex also indicates that there is a reserved space for the corresponding vertex property entries. Otherwise, if the last segment is full, the insertion operation allocates a new one, along with new arrays for each registered vertex property.

Inserting a new segment in CSR++ is as simple as appending a new pointer to the segment array. Extending this array is lightweight, given that even for large graphs such as Twitter, CSR++ only needs to copy  $\approx 3\text{MB}$  worth of pointers.

As for edge insertions, the per-vertex edge arrays use classic allocation amortization techniques for efficient edge insertions. If there is no space left to add edges, we double the size of the array through reallocation. This way we keep the size of the allocated array as a power

of two, which helps amortize the allocation costs upon possible future insertions. Naturally, CSR++ can support different growing factors than  $2\times$  to enable tuning edge insertion and memory consumption performance.

Although CSR++ efficiently supports single vertex and/or edge insertions, in practice, insertions happen in batches, e.g., inserting a set of new transactions in a financial graph. Batch insertion enables CSR++ to leverage multi-threading and reduces the cost of maintaining per-vertex edge sorting. Batch insertions are implemented with the following steps:

1. Collect an input of edges grouped by their source vertices and convert both source and destination user keys to internal keys. New vertices are inserted in CSR++ and each acquires a new internal ID. We keep this step sequential in CSR++, as it is very lightweight (see Section 4.2).
2. Sort new edges (parallel for each source vertex) then insert them in direct and reverse maps (parallel for each source vertex).
3. Sort the final edge arrays using a technique that merges two sorted arrays (i.e., the old edges and the new ones) and reallocate edge properties (parallel for each modified segment) according to the new order of edges.

**Vertex and Edge Deletion.** Deletions are not very frequent in real-life workloads. Accordingly, we develop a very lightweight protocol of logical deletions. As presented above, for vertices, setting the `length` to a negative value indicates an invalid/deleted vertex. For edges, the separate `delete_flag` indicates deletion. Of course, vertex and edge iterators are adapted to take these flags into account and disregard deleted entities. Optionally, when deleting a vertex, the list of neighbors can be destroyed. Currently, CSR++ instead restricts access to the edges if the vertex `length` is negative. Since CSR++ does not store explicit edge keys, deleting an edge requires to translate the source and destination vertex keys to internal IDs and scan over the neighbor list to locate the edge to be deleted. As already mentioned, for fast scans, CSR++ keeps the per-vertex edges sorted and performs binary searches. In case storage becomes very fragmented due to many deletions, a rather heavyweight compaction operation needs to be invoked to physically remove logically deleted entities. The cost of this operation is proportional to the cost of populating the same graph from scratch. However, we expect that this operation seldom happens in real-life deployments. Additionally, segments with no deletions can be reused as-is in the compacted graph.

### 3.3 Algorithms on Top of CSR++

CSR++ is written in C++ and is simple to use when writing graph algorithms. To iterate over vertices, CSR++ requires a nested loop to iterate over the segments then over the vertices per segment. Using parallelism APIs, such as OpenMP [4], the nested loops can be automatically collapsed and optimized. For algorithms requiring access to edges, the vertex structure implements a `get_neighbors()` method that returns its edge list.

## 4 Evaluation

In this section, we answer to the following questions regarding the performance of CSR++: How does CSR++ perform on read-only and on update workloads? How much memory does CSR++ consume on these workloads? How does CSR++ perform in comparison to other read-friendly (i.e., CSR) and update-friendly graph structures (i.e., adjacency lists and LLAMA [23])?

To this end, we compare the graph-structure configurations in Table 1, using two real-world graphs [7], LiveJournal (4.8 million vertices and 68 million edges) and Twitter (41 million vertices and 1.4 billion edges), as well as the four algorithms in Table 2 in various workload configurations. Before we present the experimental results, we describe our configuration.

■ **Table 1** Graph structures and the configurations that we use in our evaluation.

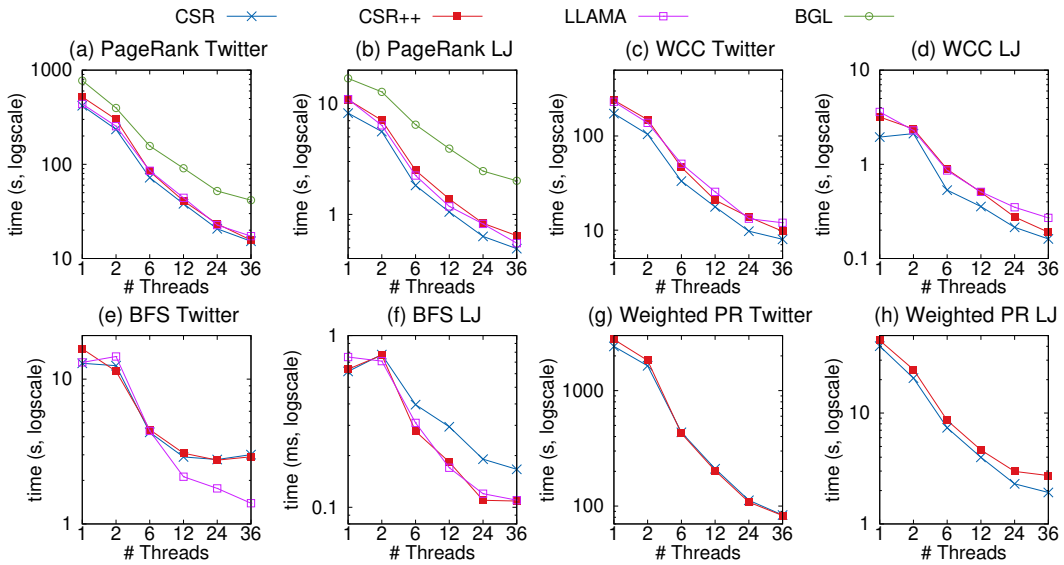
Name	Type	Configuration
CSR++	Segmentation based	Pre-allocating extra space for new edges. Deletion support enabled only on deletion workloads, in order to have fair comparison to LLAMA that does not support deletions by the default.
BGL [1]	Adjacency list	Bidirectional with default parameters.
CSR [2]	CSR	Implementation in the Green-Marl library [2].
LLAMA [3]	CSR with delta logs	Read- and space-optimized with explicit linking. The fastest overall variant of LLAMA. Deletion support enabled only on deletion workloads.

■ **Table 2** Algorithms used in our evaluation.

Algorithm	Description
PageRank	Computes ranking scores for vertices based on their incoming edges.
Weakly Connected Components (WCC)	Computes affinity of vertices within a network.
Breadth-First Search (BFS)	Traverses the graph, starting from a root vertex, visits neighbors and stores distance of vertices from the root vertex, as well as parents.
Weighted PageRank	Computes ranking scores like the original PageRank and allows a weight associated with every edge. It requires access to edge properties.

**Experimental Methodology.** For every result point, we perform five iterations and plot the median. We report the execution time as a function of the number of threads. For most analytics workloads we use CSR as a baseline. We run our benchmarks on a two-socket, 36-core machine with 384GB of RAM. Its two 2.30Ghz Intel Xeon E5-2699 v3 CPUs have 18 cores (36 hardware threads) and 32KB, 256KB and 46MB L1, L2, and LLC caches, respectively. We disable Intel TurboBoost and do not use Intel Hyper-Threading in all experiments. Both CSR++ and the other evaluated systems are implemented in C++ and compiled using GCC 4.8.2, with optimization level `-O3` and `-fopenmp` on Oracle Linux 7.3. We use the implementation of graph algorithms from Green-Marl [2].

We use the evaluated graphs as follows. For the read-only and deletion workloads, we initially load the whole graph structures. For workloads with insertions, we initially load 80% of the graph and then insert batches of different sizes, generated using the graph-split techniques used for loading and testing in the original LLAMA paper [23]. The first split contains the 80% of the graph ( $\approx 1.1$  billion edges for Twitter) that is loaded as a base graph. Then, the remaining 20% is split using a random uniform distribution over  $N$  files; we refer to  $N$  as the number of batches. Depending on the workload, we refer in figures to either the batch size (e.g., 1% corresponds to splitting the 20% in 20 batches, hence 1% of the overall graph), or the number of batches.



■ **Figure 3** Read-only performance of CSR, CSR++, LLAMA, and adjacency lists (BGL).

#### 4.1 Read-Only Workloads

We load the graph in memory and execute the evaluated algorithms. We report the execution time taken to complete each algorithm and examine how it scales with multiple threads.

Figure 3 includes the results for CSR++, CSR, BGL adjacency lists, and LLAMA. As expected, the read-only CSR provides the best performance in this workload, since with CSR, any graph access, for vertices, edges, and properties is as simple and efficient as an indexed array access. Still, CSR++ delivers performance comparable to CSR, especially in the presence of multi-threading. Over all datapoints, CSR++ is on average 15% slower than CSR, while with 36 threads, CSR++ is on average less than 10% slower than CSR.

As shown in Figure 3, we evaluate BGL adjacency lists only with PageRank. The reason for this is that the other algorithms require reverse-to-forward edge mapping, which is not supported out of the box in BGL. Still, the results of PageRank are conclusive: plain adjacency lists cannot deliver performance comparable to read-friendly structures such as CSR and CSR++. Based on these results, and for simplicity of presentation, we omit adjacency lists from the experiments in the rest of the paper.

Compared to LLAMA, CSR++ is faster for four out of the six configurations by 16% on average with 36 threads. Overall, the two systems perform within 1% of each other on average. LLAMA is faster than CSR++ for PageRank with Livejournal and for BFS on Twitter.

For Weighted PageRank, we only evaluate CSR and CSR++ and omit LLAMA because it does not support edge properties out of the box. CSR++ still performs close to CSR as shown in Figures 3h and 3i. With 36 threads, CSR++ is 1% faster than CSR on Twitter and 42% slower for Livejournal. The slowdown in Livejournal is due to the small size of the graph: with CSR’s representation, all data is served from the last-level cache, while CSR++ needs to slightly spill to main memory. These results show that the representation of edge properties in CSR++ performs comparably to CSR, especially on large graphs.

Overall, CSR++ is very fast on read-only workloads, especially in the presence of concurrency, which is the intended use case of graph analytics.

## 4.2 Updates: Vertex Insertions

Vertex insertions in CSR++ are very lightweight, mainly due to segmentation (see Section 3.2). Table 3 shows the time to insert different number of vertices on a fully loaded Twitter graph (the choice of the graph has little impact on the performance of vertex insertions in CSR++), when the graph contains either no vertex properties or 50 vertex properties. Vertex insertions are fast: With 10M insertions, inserting a vertex takes an average of 118 and 126 nanoseconds per-vertex with no and 50 properties, respectively. Vertex properties are lightweight in CSR++, as they require just one memory allocation per property per segment.

■ **Table 3** Time to add new vertices to Twitter graph in milliseconds.

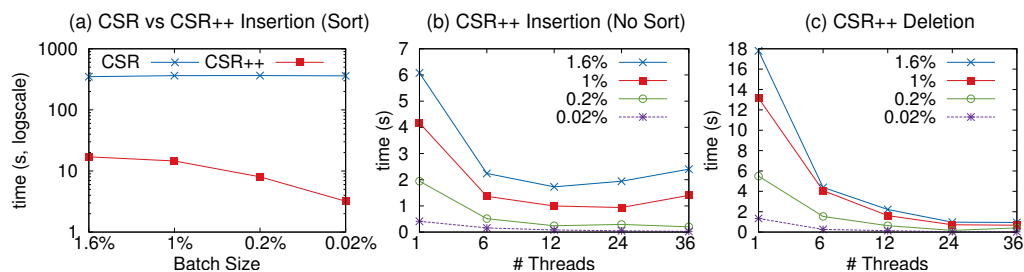
#Vertices	10K	100K	1M	10M
Time (ms) – 0 vertex properties	1.6	11	120	1188
Time (ms) – 50 vertex properties	10	32	181	1259

## 4.3 Updates: Edge Insertions

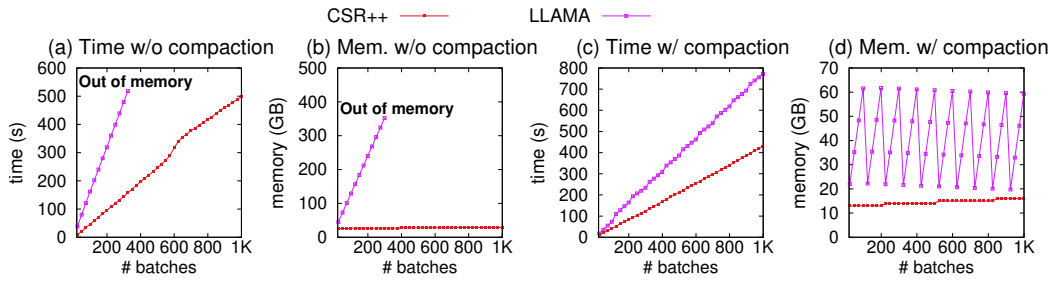
First, we evaluate the time to insert all edges of one batch in both forward and reverse structures (plus edge semi-sorting). Figure 4a shows the results. CSR++ completes this full batch insertion one order of magnitude faster than CSR. As expected, CSR completes all batch insertions in the same amount of time, regardless of the batch size. In contrast, CSR++ performs localized graph updates and thus delivers fast performance that is proportional to the batch size.

Next, we examine the scalability of edge insertions with CSR++ using the same workloads and exploiting multi-threading. The results are shown in Figure 4b. We isolate insertions by removing the edge semi-sorting that takes a significant amount of overall insertion time. CSR++ achieves good scalability for up to 12 threads. For more threads, performance does not improve, in part because of the effects of memory contention and NUMA, but mainly because of actual vertex contention: Twitter is a very skewed graph, hence many of the edge insertions land in the same high-degree vertices, hindering parallelism. Note that for these workloads, due to limited space, we only show the results with Twitter; we reach very similar conclusions with the smaller LiveJournal graph.

We further compare CSR++ to LLAMA for graph insertions. In Figure 5, we compare the edge-insertion latency and memory consumption. We apply 1000 batches of insertions (equivalent to the 0.02% workload in Figure 4), and print the memory usage and timestamp



■ **Figure 4** Graph mutations on Twitter. (a) Time to insert batches of edges of different sizes in CSR and CSR++ and sorting the edge arrays using 36 threads; (b) Time to insert different batch sizes in CSR++ without sorting using 1 to 36 threads; (c) Time to delete different batch sizes in CSR++.



■ **Figure 5** Memory consumption and batch-insertion latency of update workloads with 36 threads. (a) & (b): Comparing CSR++ and LLAMA without compaction. (c) & (d): Comparing CSR++ and LLAMA with compaction after every 100th batch insertion.

after inserting each batch. As shown in Figures 5a and 5b, the memory usage of LLAMA explodes after applying 370 batches, causing the system to run out of memory. In contrast, CSR++ consumes memory proportional to the actual graph size. Additionally, CSR++ is up to  $2.7\times$  faster in performing the insertions.

LLAMA provides a function to compact all snapshots into a single one. Figures 5c and 5d show the performance of CSR++ and LLAMA with compaction. After every 100 batches, we compact all 100 snapshots. LLAMA’s memory usage increases until compaction is invoked, but it is still higher than CSR++, even immediately after compaction. Note that the compaction method in LLAMA does not provide instructions for building the reverse edges, hence these figures show the performance of inserting only forward edges. In principle, building the reverse edges is quite more expensive than building forward edges, i.e., if the reverse operation was included, the cost of compacting would be significantly higher. Compacting 100 snapshots with only direct edges in LLAMA takes up to 40 seconds with a single thread and 5-7 seconds with 36 threads. As shown in Figure 5c, CSR++ is still consistently faster than LLAMA by a factor of approximately  $1.8\times$ .

#### 4.4 Updates: Edge Deletions

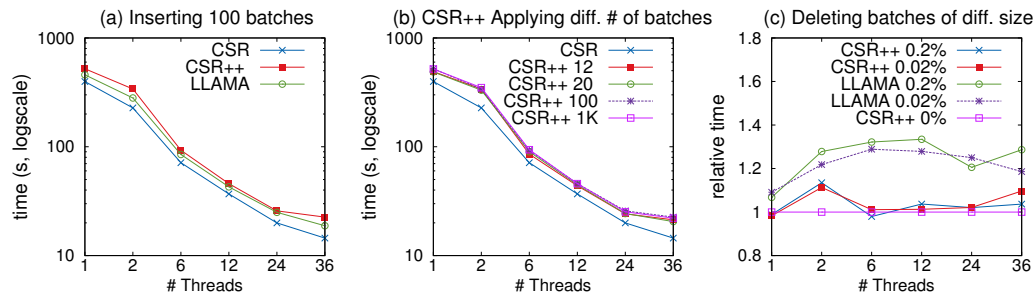
To support edge deletions, we modify our vertex and edge iterators in CSR++ to check whether a vertex or an edge is deleted, using the embedded flag in their respective structures. For LLAMA, we enable the deletion vector which similarly adds the cost of checking whether edges are deleted. Figure 4c shows the time CSR++ takes to perform edge deletions. Each data point represents the time to delete a whole batch of edges. The scalability is almost linear relative to the number of threads, and, as we increase the batch size, the effect of multithreading is more noticeable.

With a single thread, deletions are more computationally heavy, and therefore slower than insertions, as can be seen in Figures 4b and 4c. As we mention earlier, CSR++ does not store edge indices (it would be very memory consuming), which means that for every edge that is deleted, the thread needs to perform a (binary) search to find the target edge to delete logically. Note that CSR++ can “easily” also support physical deletion of edges, at the additional cost of having to reshuffle edge properties to match the new edge array.

#### 4.5 Analytics After Graph Updates

To evaluate CSR++ in a mutation context, we first load the initial 80% of the graph and simulate the insertion of a stream of updates (batches of new edges and new vertices), then we evaluate PageRank. For insertions, this workload evaluates the impact of updates on





■ **Figure 6** PageRank performance after graph updates. (a) Comparing performance of CSR, CSR++, and LLAMA after applying 100 (of size 0.2%) batches of new edges; (b) Performance of CSR++ after applying 12 (1.6%), 20 (2%), 100 (0.2%) and 1000 (0.02%) batches of new edges, CSR is used as a baseline; (c) Performance of CSR++ and LLAMA after deleting one batch of edges of different sizes.

the performance of the graph structures, e.g., for CSR++ reallocations of edge arrays and the added pointers to newly-allocated segments. For deletions, we examine the overhead of the extra conditional branch to check the deleted flags and the cost of virtual deletions.

Figure 6 shows the performance of PageRank with CSR++ and LLAMA after applying mutations to the graph. In Figure 6a, we observe that, after inserting 100 batches of new edges, the performance of CSR++ only decreases by a factor of less than  $1.25\times$  as compared to CSR, which shows the moderate overhead that is caused by the continuous reallocations of edge arrays and the copy-on-write of the indirection layer. Additionally, LLAMA is faster than CSR++ by a factor of  $1.12\times$  but consumes  $\approx 5\times$  more memory than CSR++ (see also Table 4). This is due to the 100 snapshots LLAMA stores as multi-versioning support. If we need to perform analytics on the latest version of the graph (which is the case of most real-world scenarios), the significant memory overhead of these snapshots may not be worth the minimal performance improvement. Figure 6b shows a breakdown of the performance of CSR++ when inserting different numbers of new batches: increasing the number of batches results in more reallocations and copy-on-write operations. CSR++ scales well in all cases and keeps the moderate overhead of  $\approx 1.25\times$  over CSR even after inserting 1000 batches.

Finally, Figure 6c shows the performance of CSR++ and LLAMA after deleting one batch of edges of different sizes, relative to CSR++’s performance without deletions. As we mention earlier, we modified the iterators in CSR++ to check for deletion flags in vertices and edges. We delete up to 23 million edges from the 1.47 billion total edges of Twitter, and as expected, the performance is similar to that of the baseline (i.e., without deletions). The extra conditional branches in CSR++ do not introduce considerable overhead. In case there are only few deletions, branch prediction makes sure that these deletion checks have minimal overhead, resulting in performance close to the original implementation (i.e., without deletion checks). In contrast, LLAMA’s performance significantly suffers when enabling support for deletions and makes LLAMA  $\approx 30\%$  slower than CSR++.

## 4.6 Memory Footprint

We calculate the memory footprint of Twitter and LiveJournal graphs stored in CSR, CSR++, and LLAMA (read-optimized), both just after loading them in memory and after applying different numbers of batch insertions on Twitter (Table 4).

As shown in Table 4, CSR is the most compact representation and consumes the least memory – at the cost of mutability. The memory overhead of LLAMA is small when storing one snapshot (i.e., before applying mutations), but as can be seen in the same Table 4, this

■ **Table 4** Memory footprint of different graph structures in GB in read-only workloads and after inserting a different number of batches (Twitter-x, where x is the number of batches).

Graph Structure	LiveJournal	Twitter	Twitter-12	Twitter-20	Twitter-100
CSR	0.53	11.09	11.09	11.09	11.09
CSR++ read-only	0.57	11.54	-	-	-
CSR++	0.82	16.55	16.55	16.55	16.55
LLAMA	0.58	11.56	21.66	27.03	78.00
LLAMA implicit linking	0.58	11.56	19.02	23.99	73.64

overhead increases steeply when applying batches. It is primarily due to storing different delta-snapshots of the graph for versioning. As mentioned earlier, for realistic workloads such as applying updates at a high frequency and then running analytics on recent versions of the graph, this memory overhead may lead to out-of-memory errors. As a reference, we include a second variant of LLAMA with implicit linking across snapshot versions, which trades performance for memory. The memory savings of this variant are low, however, and its performance is significantly worse (hence why our performance figures do not include it).

The default version of CSR++ has a moderate memory overhead of 33% compared to CSR, due to the pre-allocation of extra space for edge arrays. When this optimization is disabled, memory is allocated in a tight manner and CSR++ consumes closely to CSR.

## 5 Concluding Remarks

We introduced CSR++, a new concurrent graph data structure that is as fast as the fastest existing read-only graph structure, namely CSR, while enabling fast and memory-efficient in-place graph mutations. CSR++ achieves this sweet spot by combining the array-based design of CSR with the mutability of adjacency lists. In practice, CSR++ is within 10% of the performance of CSR and delivers an order of magnitude faster updates.

Future work includes using smarter synchronization mechanisms in CSR++ (such as e.g., developing lock-free protocols to avoid per-segment locking) as well as improving scalability with concurrent updates. Furthermore, we intend to explore smarter, faster, and locality-preserving memory reallocations using different memory allocators that are better suited for multithreaded applications.

---

## References

- 1 Boost Adjacency-List Documentation. [https://www.boost.org/doc/libs/1\\_67\\_0/libs/graph/doc/adjacency\\_list.html](https://www.boost.org/doc/libs/1_67_0/libs/graph/doc/adjacency_list.html).
- 2 Green-Marl Code. <https://github.com/stanford-ppl/Green-Marl>.
- 3 LLAMA Code. <https://github.com/goatdb/llama>.
- 4 OpenMP. <https://www.openmp.org>.
- 5 PGQL: Property Graph Query Language. <http://pgql-lang.org/>.
- 6 Property Graph Model. <https://github.com/tinkerpop/blueprints/wiki/Property-Graph-Model>.
- 7 SNAP (2014). Stanford Network Analysis Platform. <http://snap.stanford.edu/snap>.
- 8 SPARQL Query Language For RDF. <http://www.w3.org/TR/rdf-sparql-query/>.
- 9 Tinkerpop, Gremlin. <https://github.com/tinkerpop/gremlin/wiki>.
- 10 Tim Berners-Lee, James Hendler, Ora Lassila, et al. The Semantic Web. *Scientific american*, 284(5), 2001.

- 11 Raymond Cheng, Enhong Chen, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, and Feng Zhao. Kineograph: Taking The Pulse Of A Fast-changing And Connected World. In *EuroSys*, 2012.
- 12 Laxman Dhulipala, Guy Blelloch, and Julian Shun. Julienne: A Framework For Parallel Graph Algorithms Using Work-efficient Bucketing. In *SPAA*, 2017.
- 13 Vinicius Dias, Carlos H. C. Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. Fractal: A General-Purpose Graph Pattern Mining System. In *SIGMOD*, 2019.
- 14 David Ediger, Jason Riedy, David A. Bader, and Henning Meyerhenke. Tracking Structure of Streaming Social Networks. In *IPDPSW*, 2011.
- 15 Soukaina Firmli and Dalila Chiadmi. A Review Of Engines For Graph Storage And Mutations. In *Innovation In Information Systems And Technologies To Support Learning Research*, 2020.
- 16 Gartner. Gartner Top 10 Data And Analytics Trends For 2019. <https://www.gartner.com/smarterwithgartner/gartner-top-10-data-analytics-trends/>.
- 17 Michael Haubenschild, Manuel Then, Sungpack Hong, and Hassan Chafi. ASGraph: A Mutable Multi-versioned Graph Container With High Analytical Performance. In *GRADES*, 2016.
- 18 S. Hong, S. Depner, T. Manhardt, J. Van Der Lugt, M. Verstraaten, and H. Chafi. PGX.D: A Fast Distributed Graph Processing Engine. In *SC*, 2015.
- 19 Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-Marl: A DSL For Easy And Efficient Graph Analysis. In *ASPLOS*, 2012.
- 20 Nikolaos D. Kallimanis and Eleni Kanellou. Wait-Free Concurrent Graph Objects With Dynamic Traversals. In *OPODIS*, 2016.
- 21 Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. Graphflow: An Active Graph Database. In *SIGMOD*, 2017.
- 22 Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In *OSDI*, 2012.
- 23 P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer. LLAMA: Efficient Graph Analytics Using Large Multiversioned Arrays. In *ICDE*, 2015.
- 24 K. Madduri and D.A. Bader. Compact Graph Representations And Parallel Connectivity Algorithms For Massive Dynamic Network Analysis. In *IPDPS*, 2009.
- 25 Mugilan Mariappan and Keval Vora. GraphBolt: Dependency-Driven Synchronous Processing of Streaming Graphs. In *EuroSys*, 2019.
- 26 Daniel Mawhirter and Bo Wu. AutoMine: Harmonizing High-level Abstraction And High Performance For Graph Mining. In *SOSP*, 2019.
- 27 Neo4j. Neo4j Graph Database. <http://www.neo4j.org>.
- 28 Oracle. Parallel Graph Analytics (PGX). <https://www.oracle.com/middleware/technologies/parallel-graph-analytix.html>.
- 29 Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The Pagerank Citation Ranking: Bringing Order To The Web. Technical report, Stanford InfoLab, 1999.
- 30 Marcus Paradies, Wolfgang Lehner, and Christof Bornhövd. GRAPHITE: An Extensible Graph Traversal Framework For Relational Database Management Systems. In *SSDBM*, 2015.
- 31 Raghavan Raman, Oskar van Rest, Sungpack Hong, Zhe Wu, Hassan Chafi, and Jay Banerjee. PGX.ISO: Parallel And Efficient In-memory Engine For Subgraph Isomorphism. In *GRADES*, 2014.
- 32 Nicholas P. Roth, Vasileios Trigonakis, Sungpack Hong, Hassan Chafi, Anthony Potter, Boris Motik, and Ian Horrocks. PGX.D/Async: A Scalable Distributed Graph Pattern Matching Engine. In *GRADES*, 2017.
- 33 Sherif Sakr, Sameh Elnikety, and Yuxiong He. G-SPARQL: A Hybrid Engine For Querying Large Attributed Graphs. In *ACM CIKM*, 2012.
- 34 Martin Sevenich, Sungpack Hong, Oskar van Rest, Zhe Wu, Jayanta Banerjee, and Hassan Chafi. Using Domain-specific Languages For Analytic Graph Databases. *PVLDB*, 9(13):1257–1268, September 2016.

## 17:16 CSR++: A Fast, Scalable, Update-Friendly Graph Data Structure

- 35 Julian Shun and Guy E. Blelloch. Ligra: A Lightweight Graph Processing Framework For Shared Memory. In *PPoPP*, 2013.
- 36 Christian L. Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. NetworKit: A Tool Suite For Large-Scale Complex Network Analysis. *Network Science*, 4(4):508–530, 2016.
- 37 Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guo Tong Xie. SQLGraph: An Efficient Relational-Based Property Graph Store. In *SIGMOD*, 2015.
- 38 Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. PGQL: A Property Graph Query Language. In *GRADES*, 2016.
- 39 Brian Wheatman and Helen Xu. Packed Compressed Sparse Row: A Dynamic Graph Representation. In *HPEC*, 2018.
- 40 Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. A Distributed Graph Engine For Web Scale RDF Data. *PVLDB*, 6(4), 2013.
- 41 Kaiyuan Zhang, Rong Chen, and Haibo Chen. NUMA-Aware Graph-Structured Analytics. In *PPoPP*, 2015.

# Locally Solvable Tasks and the Limitations of Valency Arguments

Hagit Attiya 

Computer Science Department, Technion, Haifa, Israel  
hagit@cs.technion.ac.il

Armando Castañeda

Instituto de Matemáticas, UNAM, Mexico City, Mexico  
armando.castaneda@im.unam.mx

Sergio Rajsbaum 

Instituto de Matemáticas, UNAM, Mexico City, Mexico  
rajsbaum@matem.unam.mx

---

## Abstract

An elegant strategy for proving impossibility results in distributed computing was introduced in the celebrated FLP consensus impossibility proof. This strategy is *local* in nature as at each stage, one configuration of a hypothetical protocol for consensus is considered, together with future valencies of possible extensions. This proof strategy has been used in numerous situations related to consensus, leading one to wonder why it has not been used in impossibility results of two other well-known tasks: *set agreement* and *renaming*. This paper provides an explanation of why impossibility proofs of these tasks have been of a global nature. It shows that a protocol can always solve such tasks locally, in the following sense. Given a configuration and all its future valencies, if a single successor configuration is selected, then the protocol can reveal all decisions in this branch of executions, satisfying the task specification. This result is shown for both set agreement and renaming, implying that there are no local impossibility proofs for these tasks.

**2012 ACM Subject Classification** Theory of computation → Distributed computing models; Computing methodologies → Distributed algorithms; Computing methodologies → Concurrent algorithms; Theory of computation → Concurrent algorithms; Theory of computation → Distributed algorithms

**Keywords and phrases** Wait-freedom, Set agreement, Weak symmetry breaking, Impossibility proofs

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2020.18

**Related Version** A full version of the paper is available at <https://arxiv.org/abs/2011.10436>.

**Funding** Hagit Attiya: Supported by ISF grant 380/18.

Armando Castañeda: Supported by UNAM-PAPIIT project IN108720.

Sergio Rajsbaum: Supported by UNAM-PAPIIT project IN106520.

**Acknowledgements** We thank Ulrich Schmid and the reviewers for helpful comments, and Dan Alistarh, James Aspnes, Faith Ellen, Rati Gelashvili and Leqi Zhu for helpful conversations.

## 1 Introduction

An elegant strategy for proving impossibility results in distributed computing was introduced in the celebrated FLP consensus impossibility proof [17]. This strategy is *local* in nature as at each stage, one configuration of a hypothetical protocol for consensus is considered, together with its future *valencies*, namely, the decisions the protocol may reach from this configuration. To apply it, one needs to consider only the interactions of pending transitions at the configuration, and analyze their commutativity properties. This local nature makes the strategy very powerful and flexible, and has therefore been used in numerous situations related to consensus (e.g., [1, 3, 6, 7, 8, 16, 21, 24, 25, 26, 27, 29]).



© Hagit Attiya, Armando Castañeda, and Sergio Rajsbaum;  
licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 18; pp. 18:1–18:16

Leibniz International Proceedings in Informatics



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

For this reason, it would be desirable to be able to use a local strategy, in the style of FLP, to prove impossibility results for two other important tasks: *k-set agreement* [15], an extension of consensus, where processes may decide on up to  $k$  different values, and *M-renaming* [4], where processes must pick distinct names from a given namespace of size  $M$ . Existing impossibility proofs for these tasks (e.g., [5, 9, 10, 12, 22, 23, 32]) are based on topological invariant properties of final configurations of a protocol, which are *global* in nature, namely, all final configurations are analyzed together to argue that there is no protocol for the task. For consensus, these configurations are connected, in the graph-theoretic sense. For set agreement and renaming, higher-dimensional connectivity properties are proved. Researchers have investigated why only global impossibility proofs have been used for these tasks [2].

This paper provides an explanation of why the impossibility proof strategies for set agreement and renaming have been of a global nature. It shows that one could not hope to prove that set agreement and renaming are unsolvable through a local argument, since *they are solvable in a local sense*. For a configuration  $C$  of the protocol, we denote by  $\chi(C)$  all its successor configurations. In a local FLP style of argument, one selects a configuration  $C' \in \chi(C)$ , based on the valencies of the configurations in  $\chi(C)$ . The observation is that valencies can be assigned to  $\chi(C)$ , such that for any chosen configuration  $C' \in \chi(C)$ , the protocol can reveal decisions in all final configurations extending  $C'$ , such that the decisions are consistent both with the valencies and with the task specification. Intuitively, a hypothetical protocol for set agreement or renaming can “hide” its errors, if one inspects it only locally.

Intuitively, the reason that a protocol can do this for set agreement and renaming, and not for consensus, is that the consensus specification is one-dimensional in nature, so one can “corner” the protocol to reveal a configuration violating agreement (assuming the protocol terminates). Formally, it is always possible to find a *bivalent* configuration for consensus, and it is impossible to locally solve consensus from such a configuration. For set agreement and renaming, the protocol can “move” its errors around, on a higher dimensional space, without being cornered, even if the protocol declares all its valencies.

In more detail, given a hypothetical full-information protocol for either set agreement or renaming, we introduce the notion of *valency task* for set agreement and for renaming. The inputs to such a task are the configurations  $\chi(C)$  of the protocol after  $\ell$  rounds,  $\ell \geq 1$  (one round after some configuration  $C$ ). For each configuration  $C' \in \chi(C)$ , there is a valency,  $val(C')$ , specifying the outputs of the protocol on executions starting in  $C'$ . The valency task is thus defined together by both  $\chi(C)$  and the valencies. A protocol *solves the valency task locally* in  $m \geq 1$  rounds, if starting on any  $C' \in \chi(C)$ , after  $m$  rounds it produces decisions that are consistent with the task specification (either set agreement or renaming), and additionally *complete*, that is, if a value  $v \in val(C')$ , then at least one process decides  $v$  in at least one execution starting in  $C'$ . This captures the notion that the values promised by valencies are indeed decided.

We present the notions of valency task and local solvability in Section 3, and define valency tasks for set agreement and renaming. We show in Section 4 that for both valency tasks, set agreement and renaming, for any  $\ell \geq 1$ , the task is locally solvable, in one round ( $m = 1$ ) in the wait-free model. This theorem implies our main result that there are no local proofs, in the style of FLP, for set agreement and renaming, as shown in Section 5, where we present a precise notion of *local impossibility proof*. The techniques are based on combinatorial topology arguments explaining how a protocol can “hide” the inevitable mistakes it must make in some final decisions.

The setting used is a round-based wait-free model, where  $n$  asynchronous processes communicate reading and writing shared variables. Since the model is wait-free, the impossibility results are related to  $k$ -set agreement,  $k = n - 1$ , and  $M$ -renaming,  $M = 2n - 1$ . Working in a round-based model facilitates identification of consistent layers of configurations, and talking about  $\ell$ -round configurations. Considering wait-free executions allows to assume the hypothetical protocol decides always after some number of rounds,  $R$ . The significance of these specific cases and the choice of the model is further discussed in Section 6, which also explains the relation of our results to the approach of Alistarh, Aspnes, Ellen, Gelashvili and Zhu [2], the first paper that has considered this question, which showed that *extension-based* techniques do not suffice for proving the impossibility of solving set agreement.

## 2 Model of Computation and Its Topological Interpretation

The model we consider is a standard shared-memory system with  $n \geq 2$  asynchronous wait-free processes,  $P_0, \dots, P_{n-1}$ , communicating by atomically reading and writing to shared variables.

**The IIS model.** A *protocol* specifies, for each process, the steps to perform in order to solve a task. We consider an *iterated immediate snapshot* (IIS) [31] model of computation in which the protocol proceeds in a sequence of asynchronous *rounds*. In each round  $r \geq 1$ , a process performs an *immediate snapshot* (IS) operation on a clean shared array  $M[r]$ . The execution of an IS operation on  $M[r]$  is described as a sequence of *concurrency classes*, i.e., non-empty sets of processes. Each concurrency class indicates that the processes in the class first write in  $M[r]$  (in some arbitrary order) and then read all entries of  $M[r]$  (in some arbitrary order). Each process appears in exactly one concurrency class for round  $r$ , namely, executes one IS, on each memory  $M[r]$ .

An *execution* starting in  $\sigma$  is defined by a sequence of IS executions, one for each  $M[r]$ : the sequence of concurrency classes on  $M[1]$ , followed by the sequence of concurrency classes on  $M[2]$ , and so on. Since processes access a clean memory  $M[r]$  in every round  $r$ , IIS executions can be equivalently defined as a sequence of concurrency classes with the property that, for each concurrency class  $C$ , the processes in it perform the same number of IS operations in the concurrency classes preceding  $C$ . This means that all of them are poised to perform an IS operation on the same  $M[r]$ .

A *configuration* of the protocol  $\sigma = \{(P_0, v_0), \dots, (P_{n-1}, v_{n-1})\}$  consists of the local state  $v_i$  for each process  $P_i$ , during an execution. Notice that the states of the processes define the values assigned to the entries of  $M[r]$ . In an *initial configuration*  $\sigma$ , each process of  $\sigma$  is in an initial state determined by its input value (and its id), and all shared variables hold their initial value. A *partial configuration* of a configuration  $\sigma$  is a subset of  $\sigma$ .

**Tasks.** A *task*  $T = (I, O, \Delta)$  is specified by a set of input assignments  $I$  to the processes participating in an execution, a set of possible output assignments  $O$  to the participating processes, and a mapping  $\Delta : I \mapsto 2^O$  specifying the allowable outputs for each input assignment. A protocol *solves a task*  $T$  if in every execution starting in any initial configuration  $\sigma \in I$ , every participating process of  $\sigma$  decides an output value, such that the output values of the processes respect  $\Delta$  for their input values. The *safety* property is that the decisions of the processes starting with inputs  $\sigma \in I$  define an output simplex  $\tau$ , such that  $\tau \in \Delta(\sigma)$ . The *liveness* property is that the protocol is *wait-free*, namely, a process does not take an infinite number of steps without deciding.



A task is solvable in the IIS model if and only if it is solvable in the standard asynchronous read/write model [11, 18]. When one is interested only in computability (and not complexity), the protocol may be assumed to be *full-information*: a process remembers everything, and always writes all the information it has. Therefore, the protocol only needs to instruct a process when to *decide*, and on which output value.

The following tasks are defined over a domain of possible inputs  $V = \{0, 1, \dots, n-1\}$ . For proving impossibility results, it suffices to assume that a process  $P_i$  starts with input  $i$ .

► **Definition 1.** *In the  $k$ -set agreement task [15] processes decide on at most  $k$  different values, among the input values they have observed. The case where  $k = 1$  and  $V = \{0, 1\}$ , is the binary consensus task.*

► **Definition 2.** *In the  $M$ -renaming task [4] processes start with distinct values from a large domain, and decide on distinct values from a smaller domain  $\{0, \dots, M-1\}$ . In the weak symmetry breaking task [19] processes decide values in  $\{0, 1\}$ , such that not all of them decide the same value.*

If there is a protocol solving  $(2n-2)$ -renaming then there is a protocol solving weak symmetry breaking [19]. Due to its simpler structure and equivalence to  $(2n-2)$ -renaming, we study weak symmetry breaking instead of studying renaming.

**Topological Interpretation.** Since protocols preserve topological invariants of the model of computation, and these invariants, in turn, determine which tasks are solvable, it is convenient to describe protocols in the topological model of distributed computing [20].

In this model, the inputs of a task form an *input complex*  $I$ , which is a family of sets closed under containment. Each set in the family is called a *simplex*. An input simplex  $\sigma \in I$  has the form  $\sigma = \{(P_i, v_i)\}$ , for some subset of processes  $P_i$ , denoted  $ids(\sigma)$ . It indicates that process  $P_i \in ids(\sigma)$  starts with input  $v_i$ . The values  $v_i$  are taken from a universe  $V$  of possible input values. The *facets* of  $I$  are the simplexes of size  $n$ , defining the initial configurations of the system. (A *facet* is a simplex that is not contained in another simplex.) The output complex  $O$  is defined similarly.

For each input simplex  $\sigma \in I$ , a *task*  $T = (I, O, \Delta)$  specifies an output simplex  $\tau \in \Delta(\sigma)$ ,  $\tau = \{(P_i, v'_i)\}$ . This means that  $P_i$  may decide  $v'_i$ , in an execution starting with inputs defined by  $\sigma$ , where the processes observe steps by processes in  $ids(\sigma)$ .

Consider tuples of the form  $(P, view)$ , where  $P$  is in  $ids(\sigma)$  and  $view$  is the state of  $P$  after  $\ell$  rounds of communication. A configuration is a simplex, a set of such tuples, specifying the states of the processes after  $\ell$  rounds. The set of all configurations starting in  $\sigma$ , after some number of rounds  $\ell$  (including the partial configurations), defines the *protocol complex*  $\chi^\ell(\sigma)$ . The configurations of  $\chi^\ell(\sigma)$  are the simplexes of this complex. For a partial configuration  $\sigma' \subset \sigma$ ,  $\chi^\ell(\sigma')$  is the subset of  $\chi^\ell(\sigma)$  corresponding to executions where the processes of  $ids(\sigma')$  see only immediate snapshots by themselves.

In our model, the topological invariant preserved is that a full-information protocol subdivides the input complex.

The protocol complex is denoted  $\chi^\ell(\sigma)$ , since it turns out that it is the  $\ell$ -th *chromatic subdivision* of  $\sigma$ . For example, when  $n = 3$ , a configuration may be drawn as a triangle, as seen in Figure 1(left). The figure depicts the subdivision obtained after one round,  $\chi(\sigma)$ , for three processes ( $p = \text{black}$ ,  $q = \text{grey}$ ,  $r = \text{white}$ ), starting in one input simplex  $\sigma$ . It describes the sequences of concurrency classes that led to four of its simplexes. Notice that a partial configuration,  $\sigma' \subset \sigma$ ,  $|\sigma'| = 3$ , is depicted as a vertex (state of one process) or as an edge (state of two processes), contained in the triangle  $\sigma$ . The subdivision  $\chi^2(\sigma)$  is obtained by replacing each triangle  $\tau$  of  $\chi(\sigma)$ , by  $\chi(\tau)$ , and so forth.

A task  $T$  is *solvable* in  $\ell$  rounds if and only if there is a *simplicial map*  $\delta$  from the  $\ell$ -th chromatic subdivision  $\chi^\ell(I)$  to  $O$  that respects  $\Delta$ , i.e., for every  $\sigma \in I$ ,  $\delta(\chi^\ell(\sigma))$  is a subcomplex of  $\Delta(\sigma)$ . (A simplicial map sends vertices of one complex to vertices of another complex, preserving simplexes.)

If the input complex is finite (i.e., the universe  $V$  of possible input values is finite), it is well-known that there is an integer  $R$ , such that processes always decide at the end of the  $R$ -th round in a wait-free protocol. (This follows directly from König's Lemma.)

The dimension of the protocol complex, as well as the input complex, is  $n - 1$ . (The *dimension of a simplex*  $\sigma$  is  $|\sigma| - 1$ , and the *dimension of a complex* is the largest dimension of any of its simplexes.)

The *carrier*,  $\text{carr}(\tau, \chi^\ell(\sigma))$ , is the smallest  $\sigma' \subseteq \sigma$ , such that  $\tau \in \chi^\ell(\sigma')$ . In the figure, for the two edges of  $\tau$ , we have  $\text{carr}(\tau', \chi^\ell(\sigma)) = \sigma'$ , and  $\text{carr}(\tau'', \chi^\ell(\sigma)) = \sigma$ .

A *carrier map*  $\Delta : I \mapsto 2^O$  sending each input simplex  $\sigma \in I$  to a subcomplex  $\Delta(\sigma)$  of  $O$ , such that  $\sigma \subseteq \sigma'$  implies  $\Delta(\sigma) \subseteq \Delta(\sigma')$ .

### 3 Valency Tasks and Local Solvability

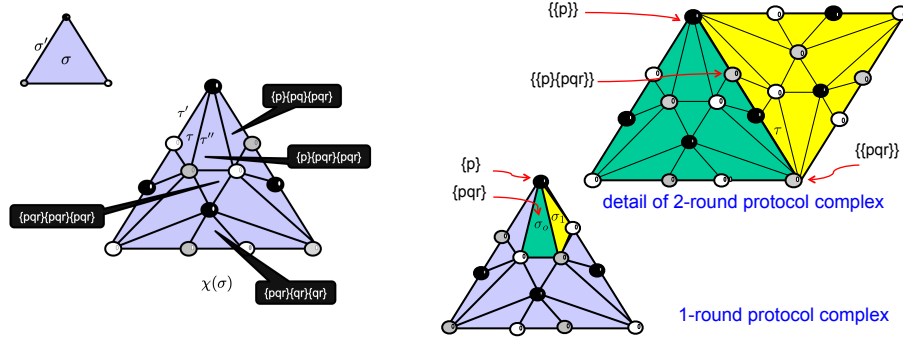
We introduce here the notions of valency task, and of locally solving such a task. Together, these notions provide the basic step in an impossibility proof in the FLP style, that will be formally defined in Section 5.

As discussed above, both for set agreement and weak symmetry breaking, one may consider, without loss of generality, a single input configuration,  $\sigma = \{(P_0, 0), \dots, (P_{n-1}, n-1)\}$ , meaning that the initial local states of the processes differ only in their ids. Thus, the input complex  $I$  consists of  $\sigma$  together with each subset of  $\sigma$ . For short, let  $\sigma = \{0, \dots, n-1\}$ , and we sometimes abuse notation and denote the input complex also by  $\sigma$ .

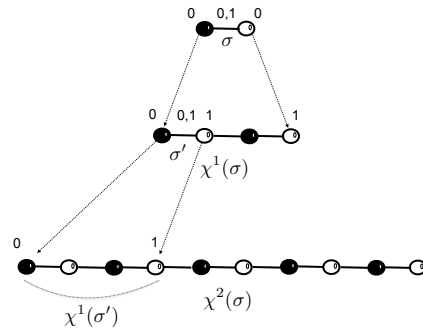
Now, assume by way of contradiction that there is a protocol  $\mathcal{P}$  solving an unsolvable task  $\mathcal{T}$  in  $R$  rounds, for some  $R \geq 1$ . Namely, the protocol complex is  $\chi^R(\sigma)$ , and each vertex  $v = (p, \text{view})$  of this complex corresponds to the state *view* of a process  $p$ , based on which,  $p$  produces an output, after executing an IS on  $M[R]$ . Solving the task means that the protocol determines a simplicial map  $\delta$ , a coloring of each vertex  $v$  of  $\chi^R(\sigma)$  with a decision value,  $\delta(v)$ , by the map  $\delta(v) = (p, \text{out})$ , in such a way that for any final configuration  $\tau \in \chi^R(\sigma)$ , the simplex of decision values  $\delta(\tau)$  belongs to  $\Delta(\sigma)$ . Since the task is unsolvable, there is no such  $\delta$ . Intuitively, a local proof demonstrates a contradiction by pinpointing a configuration  $\tau$  of the protocol complex where the decisions do not satisfy the task specification, through a local observation, as follows.

#### 3.1 Overview of the local solvability approach

Assume a protocol  $\mathcal{P}$  solving the task in  $R = \ell + m$  rounds, and consider all the configurations after  $\ell \geq 1$  rounds,  $\chi^\ell(\sigma)$ , and for each configuration  $\sigma' \in \chi^\ell(\sigma)$ , the *valencies*,  $\text{val}(\sigma')$  determined by  $\mathcal{P}$ . Namely, for each value  $v \in \text{val}(\sigma')$ , there is a final configuration  $\tau \in \chi^m(\sigma')$ , a successor of  $\sigma'$  after  $m$  rounds, such that at least one process decides on the value  $v$  in  $\tau$ . The successor configurations of  $\sigma'$  are all configurations after  $m$  additional rounds of computation by processes in  $\sigma'$ , namely, all simplexes in  $\chi^m(\sigma')$ . Figure 1 (right) depicts the case of  $\ell = m = 1$ . The successor  $\tau$  of  $\sigma'$  is reached from the initial configuration  $\sigma$  in  $\ell + m$  rounds. Given  $\chi^\ell(\sigma)$  and all the valencies of all these configurations, the impossibility argument consists of selecting one  $\sigma' \in \chi^\ell(\sigma)$ . If there are legal decisions  $\delta_{\sigma'}$  for all final configurations extending  $\sigma'$ , then the impossibility argument did not succeed in finding a contradiction, because  $\delta_{\sigma'}$  could be the map used by  $\mathcal{P}$ . This is precisely what we show for



■ Figure 1 Examples.



■ Figure 2 Consensus is not locally solvable.

set agreement and weak symmetry breaking: one can define valencies, such that for any such  $\sigma'$  there is a protocol  $\delta_{\sigma'}$  solving the task locally at  $\sigma'$ . The protocol  $\delta_{\sigma'}$  colors the vertices of  $\chi^m(\sigma')$  after executing  $m$  rounds starting in  $\sigma'$  and satisfies the task specification, and additionally, the a priori made commitments expressed by  $val(\sigma')$  (each  $val \in val(\sigma')$  is indeed decided, i.e., there is a vertex  $(p, view) \in \chi^m(\sigma')$ , with  $\delta_{\sigma'}(p, view) = (p, val)$ ). Thus, the protocol indeed preserves the valencies.

That is, an incorrect protocol can always hide the error locally. Given that the task is unsolvable, an error must exist somewhere. However, each particular configuration  $\sigma'$  inspected looks fine, and the error is moved elsewhere. We stress that this holds for every  $\ell \geq 1$  and  $m = 1$ , namely, even inspecting one round before the protocol terminates.

### 3.2 There is no Locally Solvable Valency Task for Consensus

For consensus, there is no way of defining a locally-solvable valency task. This is indeed what is expected, since there is a local impossibility proof for consensus. We show that there is no way to assign valencies, so that a protocol can hide its error. We present the case where the hypothetical protocol solves consensus in two rounds,  $\ell + m = 2$ , but the general case is analogous. (See Figure 2.)

Let  $\sigma = \{0, 1\}$  be the input edge, and the task specification  $\Delta(\{0\}) = \{(P_0, 0)\}$ ,  $\Delta(\{1\}) = \{(P_1, 1)\}$ ,  $\Delta(\{0, 1\}) = \{ \{(P_0, 0), (P_1, 0)\}, \{(P_0, 0), (P_1, 1)\} \}$ . In terms of valencies, for  $i \in \{0, 1\}$ , observe that  $val(\{i\}) = \{i\}$  (for any  $m$ ), because  $\chi^m(P_i, i)$  is the solo execution of  $P_i$  with input  $i$ , in which  $P_i$  must decide  $i$ . Thus,  $val(\sigma) = \{0, 1\}$  as  $\chi^m(P_i, i) \subset \chi^\ell(\sigma)$ .

Consider the complex  $\chi^1(\sigma)$ , which has the following edges:  $\{(P_0, \langle 0 \rangle), (P_1, \langle 0, 1 \rangle)\}$ , corresponding to the execution in which  $P_0$  goes first and then  $P_1$ ;  $\{(P_1, \langle 0, 1 \rangle), (P_0, \langle 0, 1 \rangle)\}$ , corresponding to the execution in which both processes run concurrently;  $\{(P_0, \langle 0, 1 \rangle), (P_1, \langle 1 \rangle)\}$ , corresponding to the execution in which  $P_1$  goes first and then  $P_0$ . As explained above,  $val(P_0, \langle 0 \rangle) = \{0\}$  and  $val(P_1, \langle 1 \rangle) = \{1\}$ , and the valency of any other vertex of  $\chi(\sigma)$ ,  $(P_0, \langle 0, 1 \rangle)$  and  $(P_1, \langle 0, 1 \rangle)$ , is either  $\{0\}$  or  $\{1\}$ . Thus, there must be an edge  $\sigma' = (u, v) \in \chi^1(\sigma)$  among the three edges with  $val(u) = \{0\}$  and  $val(v) = \{1\}$ , and hence  $val(\sigma') = \{0, 1\}$ . We pick such an edge  $\sigma'$  and observe that consensus is not locally solvable in  $\chi^1(\sigma')$ , i.e., the valency task with input  $\sigma'$  and outputs  $\chi^1(\sigma')$  with these valencies is not solvable. This is because any attempt to color the vertices of  $\chi^1(\sigma')$ , with one endpoint of the path colored 0 and the other colored 1, will produce an edge  $\tau$  whose vertices have different colors, violating the agreement requirement of consensus.

We have seen that for consensus (1-set agreement) it is impossible to define a valency task that is locally solvable. In Sections 3.3 and 3.4 we show how to specify valency tasks for set agreement and weak symmetry breaking that *are* locally solvable, and in Section 4 we describe protocols that solve them.

### 3.3 Valency Tasks and Local Solvability for Set Agreement

Consider now the unique input simplex  $\sigma = \{0, \dots, n-1\}$  for  $k$ -set agreement,  $k = n-1$ . Processes decide values from  $\sigma$  that they have seen, and such that at most  $n-1$  different values are decided in an execution.

Following topology terminology, in the rest of the paper configurations are called simplexes. First, recall that for a simplex  $\tau \in \chi^\ell(\sigma)$ , the *carrier* of  $\tau$  in  $\chi^\ell(\sigma)$  is the smallest face  $\sigma' \subseteq \sigma$ , such that  $\tau \in \chi^\ell(\sigma')$ . From an operational perspective,  $carr(\tau, \chi^\ell(\sigma))$  identifies the set of processes seen in the  $\ell$ -round IIS execution that ends at configuration  $\tau$ .

The goal is to define, for each  $\ell$ , a *set agreement valency task*  $\mathcal{T} = \langle \chi^\ell(\sigma), \sigma, val \rangle$ . This is a task that respects the set agreement specification: a decided value should have been seen, namely, a process deciding  $v$  must have  $v$  in its view. Indeed, agreement tasks such as consensus and set agreement are specializations of a *validity* task [14], where this is the only requirement.

More formally, in a valency task for set agreement, for every simplex  $\tau \in \chi^\ell(\sigma)$ ,  $val(\tau) \subseteq carr(\tau, \chi^\ell(\sigma))$ . The set of inputs of  $\mathcal{T}$  are the configurations at round  $\ell$ , namely  $\chi^\ell(\sigma)$ . For each configuration  $\tau \in \chi^\ell(\sigma)$ , the set of possible decisions  $val(\tau)$  is a non-empty subset of  $\sigma$  (this is the standard hypothesis of Sperner's lemma). Notice that  $val$  can be formally defined as a carrier map.<sup>1</sup> The following is a particular set agreement valency task.

► **Definition 3** (Locally solvable set agreement valency task). *For every integer  $\ell \geq 1$ , let  $\mathcal{T} = \langle \chi^\ell(\sigma), \sigma, val \rangle$ , where  $val$  is the carrier map defined by*

1. *If  $|\tau| \leq n-2$ , then  $val(\tau) = ids(\tau)$ ,*
2. *else  $val(\tau) = carr(\tau, \chi^\ell(\sigma))$ .*

In the notion of *local solvability* of valency-tasks, we ask for a protocol that solves  $\mathcal{T}$  in  $m$  rounds, namely a decision map  $c_\tau : \chi^{\ell+m}(\sigma) \rightarrow \sigma$  that respects  $\mathcal{T}$ . Thus,  $c_\tau$  is a *global* solution to  $\mathcal{T}$ , but the  $k$ -set agreement task is solved only *locally* at  $\tau$ :  $c_\tau$  is determined by a specific input simplex  $\tau \in \chi^\ell(\sigma)$ , and  $c_\tau(\chi^m(\tau))$  does not have any simplex with  $k+1$

<sup>1</sup> Formally, the corresponding task specification  $\Delta$ , for  $\Delta(\tau)$ , consists of all output simplexes labeled by output values from  $val(\tau)$ .

decisions. Of course,  $c_\tau$  does not globally solve  $k$ -set agreement because indeed  $c_\tau(\chi^{\ell+m}(\sigma))$  is a Sperner's coloring and has at least one simplex colored with  $n$  different decisions, by Sperner's lemma [33]. Recall that a Sperner coloring  $c : \chi^{\ell+m}(\sigma) \rightarrow \sigma$  is a simplicial map such that  $c(v) \in \text{carr}(\tau, \chi^{\ell+m}(\sigma))$ , for every vertex  $v$  of  $\chi^{\ell+m}(\sigma)$ .

We have that if  $c_\tau$  solves  $\mathcal{T}$  in  $m$  rounds, for each  $\tau \in \chi^\ell(\sigma)$  and all configurations after  $m$  rounds,  $\chi^{\ell+m}(\tau)$ , it should hold that  $c_\tau$  is *consistent*, i.e.,  $c_\tau(\chi^{\ell+m}(\tau)) \subseteq \text{val}(\tau)$ . We require that  $c_\tau$  is additionally *complete*, meaning that every value committed by the valencies, is indeed decided, namely,  $c_\tau(\chi^{\ell+m}(\tau)) = \text{val}(\tau)$ .

► **Definition 4** (Local solvability of  $k$ -set agreement). *We say that a set agreement valency task  $\mathcal{T} = \langle \chi^\ell(\sigma), \sigma, \text{val} \rangle$  is  $k$ -locally solvable in  $m \geq 1$  rounds if for every input simplex  $\tau \in \chi^\ell(\sigma)$  there is a decision simplicial map  $c_\tau : \chi^{\ell+m}(\sigma) \rightarrow \sigma$  that is consistent and complete w.r.t.  $\mathcal{T}$  and  $c_\tau(\chi^m(\tau))$  does not have simplexes with more than  $k$  distinct decisions at its vertices.*

We stress that local solvability allows  $c_\tau$  (which depends on  $\tau$ ) to have simplexes not in  $\chi^m(\tau)$  with more than  $k$  distinct decisions, as it requires that  $c_\tau$  solves  $k$ -set agreement only in  $\chi^m(\tau)$ . Although it is unavoidable that there are simplexes with more than  $k$  distinct decisions *somewhere* (due to the  $k$ -set agreement impossibility), local solvability does not require that the task is globally unsolvable. Indeed, while we prove (Section 4.1) that the valency task for set agreement is  $(n-1)$ -locally solvable in a single round, we do not prove it is globally unsolvable.

### 3.4 Valency Tasks and Local Solvability for Weak Symmetry Breaking

The weak symmetry breaking task with unique input  $(n-1)$ -simplex  $\sigma = \{0, \dots, n-1\}$  requires that the binary output coloring on the boundary of  $\chi^\ell(\sigma)$  has the next symmetry property (assuming the protocol terminates in  $\ell$  rounds) on the vertices  $V(\chi^\ell(\sigma))$ , e.g. [12, 13, 23]:

► **Definition 5** (Symmetric binary coloring). *A symmetric binary coloring of  $\chi^\ell(\sigma)$  is a simplicial map  $b : V(\chi^\ell(\sigma)) \rightarrow \{0, 1\}$  satisfying that, for any two distinct proper faces  $\sigma', \sigma''$  of  $\sigma$  of the same dimension,  $v \in V(\chi^\ell(\sigma'))$  and  $\phi(v) \in V(\chi^\ell(\sigma''))$  have the same binary color, i.e.  $b(v) = b(\phi(v))$ , where  $\phi$  is the simplicial bijection between  $V(\chi^\ell(\sigma'))$  and  $V(\chi^\ell(\sigma''))$  that maps vertices preserving order, namely, vertices with the smallest id in  $\text{ids}(\sigma')$  to vertices with the smallest id in  $\text{ids}(\sigma'')$ , vertices with the second smallest id in  $\text{ids}(\sigma')$  to vertices with the second smallest id in  $\text{ids}(\sigma'')$ , and so on.*

We remark that a weak symmetry breaking protocol can be transformed into a *comparison-based* protocol, in which processes only perform comparisons between inputs. Thus, actual input values are irrelevant, and only the relative order among them matters. In inputless weak symmetry breaking,  $i \in \sigma$  denotes the process with  $i$ -th input, in ascending order.

Output decisions in weak symmetry breaking are binary, hence in valency tasks for weak symmetry breaking the carrier map  $\text{val}$  goes from  $\chi^\ell(\sigma)$  to  $\{0, 1\}$ , the complex with a single edge, and its vertices. Since  $\text{val}$  models the valencies of a hypothetical protocol for weak symmetry breaking, the valencies must be symmetric on the boundary; this is the only requirement  $\text{val}$  must satisfy. The following is a particular weak symmetry breaking valency task, where it is not hard to check that  $\text{val}$  is indeed a carrier map.

► **Definition 6** (Locally solvable weak symmetry breaking valency task). *For every  $\ell \geq 1$ , let  $\mathcal{T} = \langle \chi^\ell(\sigma), \{0, 1\}, \text{val} \rangle$  where  $\text{val}$  is the carrier map defined by*

1. *If  $\dim(\tau) \leq n-3$ , then  $\text{val}(\tau) = \{1\}$ .*
2. *Otherwise,  $\text{val}(\tau) = \{0, 1\}$ .*

Analogous to set agreement, if a symmetric binary coloring  $b_\tau : V(\chi^{\ell+m}(\sigma)) \rightarrow \{0, 1\}$  solves  $\mathcal{T}$  in  $m$  rounds then it respects *val*, or is *consistent* with  $\mathcal{T}$ . This means that for every input simplex  $\tau \in \chi^\ell(\sigma)$ ,  $b_\tau(\chi^{\ell+m}(\tau)) \subseteq \text{val}(\tau)$ . We also require that it is *complete*, i.e.,  $b_\tau(\chi^{\ell+m}(\tau)) = \text{val}(\tau)$ .

It has been shown [9, 12] that if  $\dim(\sigma) + 1$  is a prime power, then  $b(\chi^\ell(\sigma))$  has at least one *monochromatic* simplex (i.e. with all its vertices having the same binary color) of dimension  $\dim(\sigma)$ , which implies the impossibility of weak symmetry breaking; those monochromatic simplexes are the errors that  $b$  makes, however,  $b$  is able to hide them locally: for the specified input simplex  $\tau \in \chi^\ell(\sigma)$ ,  $b(\chi^m(\tau))$  does not have monochromatic simplexes of dimension  $\dim(\sigma)$ .

► **Definition 7** (Local solvability of weak symmetry breaking). *We say that a weak symmetry breaking valency task  $\mathcal{T} = \langle \chi^\ell(\sigma), \{0, 1\}, \text{val} \rangle$  is locally solvable in  $m \geq 1$  rounds if for every input simplex  $\tau \in \chi^\ell(\sigma)$  there is a symmetric binary decision map  $b_\tau : \chi^{\ell+m}(\sigma) \rightarrow \{0, 1\}$  (which is on function of  $\tau$ ) that is consistent and complete w.r.t.  $\mathcal{T}$  and  $b_\tau(\chi^m(\tau))$  does not have monochromatic simplexes of dimension  $\dim(\sigma)$ .*

In the next section, we prove that the weak symmetry breaking valency task (Definition 6) is locally solvable in one round. This result is trivial when  $\dim(\sigma) + 1$  is not a prime power because in those cases weak symmetry breaking is indeed solvable [13], and hence, there is a symmetric binary coloring with no monochromatic simplexes (i.e., without errors). The interesting case is when weak symmetry breaking is not solvable and unavoidable errors need to be hidden.

## 4 Solving Valency Tasks

This section contains the proof of Theorems 8 and 10, stating that the set agreement and weak symmetry breaking valency tasks defined in the previous section, Definitions 3 and 6, are locally solvable in one round.

### 4.1 Set Agreement

The following theorem shows that the valency tasks for set agreement defined in the previous section are locally solvable.

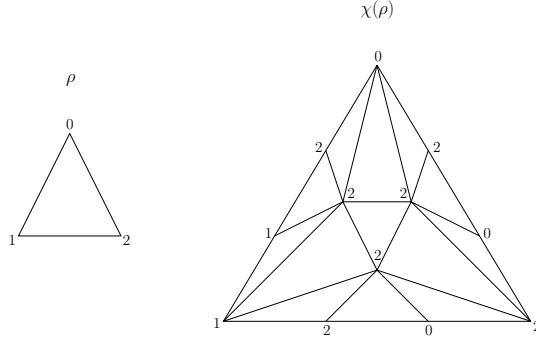
► **Theorem 8.** *For any  $n \geq 3$  and  $\ell \geq 1$ , the set agreement valency task  $\mathcal{T} = \langle \chi^\ell(\sigma), \sigma, \text{val} \rangle$  in Definition 3 is  $(n - 1)$ -locally solvable in one round.*

The proof of Theorem 8 relies on the following lemma, regarding vertex colorings of the first standard chromatic subdivision. Roughly speaking, the lemma identifies colorings that, to some extent, satisfy the properties of a Sperner coloring, but without simplexes with  $n$  different decisions. Figure 3 presents an example of these colorings.

► **Lemma 9.** *Consider the  $(n - 1)$ -dimensional simplex  $\rho = \{0, \dots, n - 1\}$  with  $n \geq 3$ . There is a coloring (simplicial map)  $c : \chi(\rho) \rightarrow \rho$  such that:*

1. *for every  $\rho' \subset \rho$  with  $\dim(\rho') \leq n - 3$ ,  $c(\chi(\rho')) = \rho'$ ,*
2. *one of the following holds:*
  - a. *for every  $(n - 2)$ -dimensional face  $\rho' \subseteq \rho$ ,  $c(\chi(\rho')) = \rho$ ,*
  - b. *for a chosen  $(n - 2)$ -dimensional  $\rho'' \subset \rho$ ,  $c(\chi(\rho'')) = \rho''$ , and for every other  $(n - 2)$ -dimensional face  $\rho' \subseteq \rho$ ,  $c(\chi(\rho')) = \rho$ ,*
3.  *$c(\chi(\rho)) = \rho$  and there is no fully colored  $(n - 1)$ -simplex in  $c(\chi(\rho))$ .*





■ **Figure 3** Example for Lemma 9. Since every vertex  $v$  of  $\rho$  has dimension  $n - 3$ , the only vertex in  $\chi(v)$  has color  $v$ , implying (1). The coloring satisfies requirement (2.b) of the lemma: for the  $(n - 2)$ -face  $\{0, 2\}$  of  $\rho$ , all vertices in  $\chi(\{0, 2\})$  have colors in  $\{0, 2\}$ , while for any other  $(n - 2)$ -face  $\rho'$  of  $\rho$ , every vertex in  $\chi(\rho')$  has a color in  $\rho$ . Finally,  $\chi(\rho)$  has no  $(n - 1)$ -simplex with the three colors at its vertices, implying (3).

For every simplex  $\tau \in \chi^\ell(\sigma)$ , let  $ids(\tau)$  be the simplex containing the first entries of the vertices in  $\tau$  (recall that each vertex of  $\chi^\ell(\sigma)$  is a pair  $(v, view)$  where  $v$  is the id of a process and  $view$  is its view after  $\ell$  rounds); note that  $ids(\tau)$  is a  $dim(\tau)$ -face of  $\sigma$ , and  $ids(\tau) \subseteq carr(\tau, \sigma, \chi^\ell)$ , since  $\chi^\ell(\sigma)$  is a chromatic subdivision of  $\sigma$ .

**Proof of Theorem 8.** We prove now that set agreement valency task  $\mathcal{T} = \langle \chi^\ell(\sigma), \sigma, val \rangle$  is  $(n - 1)$ -locally solvable in one round. To do so, we define a Sperner coloring  $c_\tau$  of  $\chi^{\ell+1}(\sigma)$  that is consistent and complete w.r.t. the task and has no fully colored  $(n - 1)$ -simplexes in  $c_\tau(\chi(\tau))$ , for any input simplex  $\tau \in \chi^\ell(\sigma)$ . We focus on the case when  $|\tau| = n$  because for any simplex  $\tau'$  of a smaller dimension, we can just pick any  $\tau$  containing  $\tau'$ , and set  $c_{\tau'}$  to  $c_\tau$  restricted to  $\chi(\tau')$ , i.e.  $c_\tau|_{\chi(\tau')}$ .

Thus, for the rest of the proof fix an  $(n - 1)$ -dimensional simplex of  $\tau \in \chi^\ell(\sigma)$ . We define a Sperner coloring  $c_\tau$  that is consistent with  $val$  and has no fully colored  $(n - 1)$ -simplexes in  $\chi(\tau)$ . First, we use Lemma 9 to define  $c_\tau$  restricted to  $\chi(\tau)$ , i.e.  $c_\tau|_{\chi(\tau)}$ , and then extend the coloring to all vertices in  $\chi^{\ell+1}(\sigma)$ , to finally obtain  $c_\tau$ .

Let  $\rho = ids(\tau)$ . Note that  $\rho = \sigma$  but for clarity we use  $\rho$ .  $ids$ 's naturally induce a bijection between vertices of  $\rho$  and  $\tau$ , and  $\chi(\rho)$  and  $\chi(\tau)$ , respectively, hence any coloring (simplicial map)  $\chi(\rho) \rightarrow \rho$  induces a coloring  $\chi(\tau) \rightarrow ids(\tau)$ . Below, when we use Lemma 9 applied to  $\rho = ids(\tau)$ , we can speak about faces of  $\tau$  instead of faces of  $\rho$ .

Observe that either for every  $(n - 2)$ -face  $\tau'$  of  $\tau$ ,  $carr(\tau', \sigma, \chi^\ell) = \sigma$ , or for one  $(n - 2)$ -face  $\tau''$  of  $\tau$ ,  $carr(\tau', \chi^\ell) = ids(\tau'')$  and for every other  $(n - 2)$ -face  $\tau'$  of  $\tau$ ,  $carr(\tau', \sigma, \chi^\ell) = \sigma$ . Intuitively,  $\tau'$  is “inside”  $\chi^\ell(\sigma)$  or only one  $(n - 2)$ -face of  $\tau'$  “touches” the boundary of  $\chi^\ell(\sigma)$  (see Figure 3). We set  $c_\tau|_{\chi(\tau)}$  using a coloring of  $\chi(\tau)$  in Lemma 9, as follows. In the former case,  $c_\tau|_{\chi(\tau)}$  is obtained with a coloring of  $\chi(\tau)$ , as in Case (2.a) of Lemma 9, while in the latter case, is obtained with a coloring as in case (2.b), where  $\tau''$  is the chosen face in that case of the lemma.

We argue that Lemma 9 and the definition of  $val$  implies that for any face  $\tau'$  of  $\tau$ , it holds that  $c_\tau(\chi(\tau')) = val(\tau')$ , which is good because we want  $c_\tau$  to be consistent and complete w.r.t.  $val$ . If  $dim(\tau') \leq n - 3$ , then  $val(\tau') = ids(\tau')$ , by definition of  $val$ , and from Lemma 9(1), we know that  $c_\tau(\chi(\tau')) = ids(\tau')$ . Also, by definition of  $val$ , if  $dim(\tau') > n - 3$ , then  $val(\tau') = carr(\tau', \sigma, \chi^\ell)$ . Note that if  $dim(\tau') = n - 1$  (hence  $\tau' = \tau$ ), then  $val(\tau') = \sigma$ , and  $c_\tau(\chi(\tau')) = val(\tau')$ , by Lemma 9(3). The subcase that remains to be shown is when  $dim(\rho) = n - 2$ . Again, if  $val(\tau') = \sigma$ ,  $c_\tau(\chi(\tau')) = val(\tau')$ , by Lemma 9(3). Thus, consider



the case  $val(\tau') = carr(\tau', \sigma, \chi^\ell) \neq \sigma$ . Observe that this can only happen when  $\tau'$  is at the boundary of  $\chi^\ell(\sigma)$ , and hence  $carr(\tau', \sigma, \chi^\ell) = ids(\tau')$ . By Lemma 9(2.b),  $c_\tau(\chi(\tau')) = ids(\tau')$  ( $\tau'$  was the chosen  $(n-2)$ -face of  $\tau$  in case (2.b) of Lemma 9 when defining  $c_\tau$  on  $\chi(\tau)$ ).

We now extend the coloring  $c_\tau$  in two steps. First, for any vertex  $v \in \chi^\ell(\sigma)$  that does not belong to  $\chi(\tau)$ , we first set  $c_\tau(v) = id(v)$ . Thus, for any input simplex  $\lambda \in \chi^\ell(\sigma)$  that does not intersect  $\tau$ , we have that  $c_\tau(\chi(\lambda)) = ids(\lambda)$ . That is fine if  $dim(\lambda) \neq n-2$ , or  $dim(\lambda) = n-2$  and  $carr(\lambda, \sigma, \chi^\ell) \neq \sigma$ , because in such cases  $c_\tau(\chi(\lambda)) = val(\lambda)$ , by definition of  $val$ .

But if  $dim(\lambda) = n-2$  and  $carr(\lambda, \sigma, \chi^\ell) = \sigma$ , then  $c_\tau(\chi(\lambda)) = ids(\lambda) \subset val(\lambda) = \sigma$ , and then in this case  $c_\tau$  is not complete. Note that the proper contention is because there is no vertex in  $\chi(\lambda)$  that is mapped to the unique vertex in  $\sigma \setminus ids(\lambda)$ . To solve this issue, for every such input simplex  $\lambda \in \chi^\ell(\sigma)$ , we pick one vertex  $v \in \chi(\lambda)$  with  $carr(v, \lambda, \chi) = \lambda$  (which belongs to the “central”  $(n-2)$ -simplex of  $\chi(\lambda)$ ) and set  $c_\tau(v)$  to the unique vertex in  $\sigma \setminus ids(\lambda)$ . Therefore, we now have that  $c_\tau(\chi(\lambda)) = val(\lambda) = \sigma$ .

To fully prove that  $c_\tau$  is consistent and complete w.r.t.  $val$ , the only case that remains is of an input simplex  $\lambda$  that intersects  $\tau$  but is not one of its faces. Let  $\lambda'$  and  $\tau'$  be the proper faces of  $\lambda$  and  $\tau$  such that  $\lambda = \lambda' \cup \tau'$ . We already know that  $c_\tau(\chi(\lambda')) = val(\lambda')$  and  $c_\tau(\chi(\tau')) = val(\tau')$ . If  $dim(\lambda' \cup \tau') \leq n-3$ , the definition of  $val$  implies that  $val(\lambda \cup \tau') = val(\lambda') \cup val(\tau')$ , hence  $c_\tau(\chi(\lambda' \cup \tau')) = val(\lambda \cup \tau')$ . If  $dim(\lambda' \cup \tau') = n-1$ , then it must be that  $val(\lambda \cup \tau') = \sigma$ , from the definition of  $val$ , and then clearly  $c_\tau(\chi(\lambda' \cup \tau')) = val(\lambda \cup \tau')$ , by construction. If  $dim(\lambda' \cup \tau') \geq n-2$ , we have two cases,  $val(\lambda' \cup \tau')$  is either  $ids(\lambda' \cup \tau')$  or  $\sigma$ ; in any case, the very definition of  $c_\tau$  implies that  $c_\tau(\chi(\lambda' \cup \tau')) = val(\lambda \cup \tau')$ .

Therefore, so far we have a coloring  $c_\tau$  that is consistent and complete w.r.t.  $val$  and  $c_\tau(\chi(\tau))$  has no fully colored  $(n-1)$ -simplexes (since we defined  $c_\tau(\chi(\tau))$  using Lemma 9). To finally conclude that  $(\chi^\ell(\sigma), \sigma, val)$  is locally solvable in one round, we argue  $c_\tau$  is a Sperner coloring, which essentially follows because  $val$  is a Sperner-valency coloring and  $c_\tau$  is consistent and complete w.r.t.  $val$ . To prove the claim in detail, consider any vertex  $v \in \chi^\ell(\sigma)$ . If  $v \notin \chi(\tau)$ , then  $c_\tau(v) = id(v) \in carr(v, \sigma, \chi^\ell)$ . Otherwise, let  $\tau' = carr(v, \tau, \chi)$ . Note that  $ids(\tau') \subseteq carr(v, \sigma, \chi^\ell)$ . It follows from Lemma 9 that  $c_\tau(\chi(\tau'))$  is either  $ids(\tau')$  or  $\sigma$ . If  $c_\tau(\chi(\tau')) = ids(\tau')$  then  $c_\tau(v) \in carr(v, \sigma, \chi^\ell)$ . For the remaining case, note that  $c_\tau(\chi(\tau')) = \sigma$  only if  $dim(\tau') = n-1$  (hence  $carr(\tau', \sigma, \chi^\ell) = \sigma$ ), or  $dim(\tau') = n-2$  and  $carr(\tau', \sigma, \chi^\ell) = \sigma$  (i.e.  $\tau'$  is not the chosen  $(n-2)$ -face of  $\tau$  in the case (2.b) of Lemma 9); in either case we have that  $c_\tau(v) \in carr(v, \sigma, \chi^\ell)$ . We conclude that  $c_\tau$  is a Sperner coloring. ◀

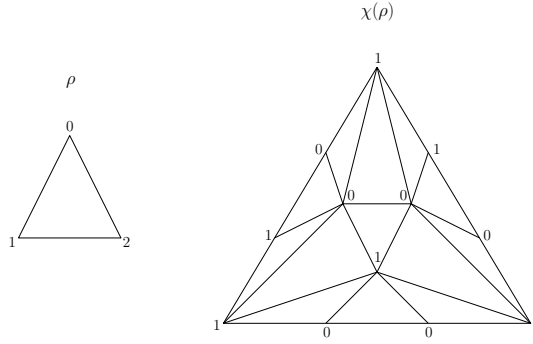
## 4.2 Weak Symmetry Breaking

► **Theorem 10.** *For any  $n \geq 3$  and  $\ell \geq 1$ , the weak symmetry breaking valency task  $\mathcal{T} = (\chi^\ell(\sigma), \sigma, val)$  in Definition 6 is locally solvable in one round.*

The proof of Theorem 10 is similar in structure to the proof for set agreement in the previous section. It relies on Lemma 11 below to produce binary colorings that are almost symmetric on the boundary and do not have monochromatic  $dim(\sigma)$ -simplexes. Figure 4 shows an example of such a coloring. In the proof of Theorem 10, we use these binary colorings to locally solve symmetric binary-valency tasks.

► **Lemma 11.** *Consider the  $(n-1)$ -dimensional simplex  $\rho = \{0, \dots, n-1\}$  with  $n \geq 3$ . There is a binary coloring (simplicial map)  $b : \chi(\rho) \rightarrow \{0, 1\}$  such that:*

1. *for every  $\rho' \subset \rho$  with  $dim(\rho') \leq n-3$ ,  $b(\chi(\rho')) = \{1\}$ ,*
2. *for every  $(n-2)$ -dimensional face  $\rho' \subseteq \rho$ ,  $b(\chi(\rho')) = \{0, 1\}$ ,*
3.  *$b(\chi(\rho)) = \{0, 1\}$  and there is no monochromatic  $(n-1)$ -simplex in  $b(\chi(\rho))$ .*



■ **Figure 4** Example for Lemma 11. Since every vertex  $v$  of  $\rho$  has dimension  $n - 3$ , the only vertex in  $\chi(v)$  has color 1, implying (1). For every  $(n - 2)$ -face  $\rho'$  of  $\rho$ , there are vertices in  $\chi(\{0, 2\})$  with color 0 and 1, implying (2). Finally,  $\chi(\rho)$  has no monochromatic  $(n - 1)$ -simplex, implying (3).

**Proof of Theorem 10.** We now show that the weak symmetry breaking valency task  $\mathcal{T} = \langle \chi^\ell(\sigma), \sigma, \text{val} \rangle$  is locally solvable in one round. We need to show that for every input simplex  $\tau \in \chi^\ell(\sigma)$ , we define a symmetric binary coloring  $b_\tau$  of  $\chi^{\ell+1}(\sigma)$  that is consistent and complete w.r.t.  $\text{val}$  and has no monochromatic  $(n - 1)$ -simplexes in  $b_\tau(\chi(\tau))$ . We focus on the case is when  $\tau$  is of dimension  $n - 1$  because for any simplex  $\tau'$  of a smaller dimension, we can just pick any  $\tau$  containing  $\tau'$ , and set  $b_{\tau'}$  to  $b_\tau$  restricted to  $\chi(\tau')$ , i.e.  $b_\tau|_{\chi(\tau')}$ .

For the rest of the proof fix an  $(n - 1)$ -dimensional simplex of  $\tau \in \chi^\ell(\sigma)$ . We define a symmetric binary coloring  $b_\tau$  that is consistent and complete w.r.t.  $\text{val}$  and has no monochromatic  $(n - 1)$ -simplexes in  $\chi(\tau)$ . First, we use Lemma 11 to define  $b_\tau$  restricted to  $\chi(\tau)$ , i.e.  $b_\tau|_{\chi(\tau)}$ , and then extend the coloring to all vertices in  $\chi^{\ell+1}(\sigma)$ , to finally obtain  $b_\tau$ .

Let  $\rho = \text{id}_s(\tau)$ . Note that  $\rho = \sigma$  but for clarity we use  $\rho$ .  $\text{id}$ 's naturally induce a bijection between  $\rho$  and  $\tau$ , and  $\chi(\rho)$  and  $\chi(\tau)$ , hence any coloring (simplicial map)  $\chi(\rho) \rightarrow \rho$  induces a coloring  $\chi(\tau) \rightarrow \text{id}_s(\tau)$ . Below, when we use Lemma 11 applied to  $\rho = \text{id}_s(\tau)$ , we can speak about faces of  $\tau$  instead of faces of  $\rho$ .

First, we set  $b_\tau|_{\chi(\tau)}$  using a coloring of  $\chi(\tau)$  in Lemma 11. We have that  $b_\tau|_{\chi(\tau)}$  is consistent and complete with respect to  $\text{val}$ : for every face  $\tau'$  of  $\tau$ , if  $\dim(\tau') \leq n - 3$ ,  $\text{val}(\tau') = \{1\}$ , by definition of  $\text{val}$ , and  $b_\tau(\chi(\tau')) = \{1\}$ , by Lemma 11(1); and if  $\dim(\tau') \geq n - 2$ ,  $\text{val}(\tau') = \{0, 1\}$ , by definition of  $\text{val}$ , and  $b_\tau(\chi(\tau')) = \{0, 1\}$ , by Lemma 11(2-3).

We extend  $b_\tau$  in two steps. In the first step, we pick any vertex  $v \in \chi^{\ell+1}(\sigma)$  that does not belong to  $\chi(\tau)$  (which is uncolored yet). If there are faces  $\sigma', \sigma''$  of  $\sigma$  with the same dimension such that  $v \in \chi^{\ell+1}(\sigma')$ ,  $\phi(v) \in \chi^{\ell+1}(\sigma'')$  and  $\phi(v) \in \chi(\tau)$ , where  $\phi$  is the simplicial bijection between  $\chi^{\ell+1}(\sigma')$  and  $\chi^{\ell+1}(\sigma'')$  that maps vertices preserving order, then set  $b_\tau(v) = b_\tau(\phi(v))$ ; otherwise, set  $b_\tau(v) = 1$ . In words: if  $\chi(\tau)$  “touches” the boundary of  $\chi^{\ell+1}(\sigma)$ , we replicate that “part” of the coloring in its symmetric “counterparts” in the boundary. Observe that  $b_\tau$  is well defined because, since  $\ell \geq 1$ , there are no two vertices of  $u, v \in \chi(\tau)$  such that there are two distinct faces  $\sigma', \sigma''$  of  $\sigma$  of same dimension such that  $u \in \chi^{\ell+1}(\sigma')$  and  $v \in \chi^{\ell+1}(\sigma'')$ ; intuitively,  $\chi(\tau)$  can “touch” either  $\chi^{\ell+1}(\sigma')$  or  $\chi^{\ell+1}(\sigma'')$  but not both. Note that  $b_\tau$  is symmetric.

It is not hard to see that for any input simplex  $\lambda \in \chi^\ell(\sigma)$  with  $\dim(\lambda) \leq n - 3$ ,  $b_\tau(\chi(\lambda)) = \text{val}(\lambda)$ . First, if  $\lambda$  is a face of  $\tau$ , we have already saw that this is true. Second, if  $\lambda$  is a face of  $\tau$ , then  $b_\tau(\chi(\lambda)) = \{1\}$  because even if a vertex  $v \in \chi(\lambda)$  is in the boundary of  $\chi^{\ell+1}(\sigma)$  and gets its color from a vertex  $u \in \chi(\tau)$  (i.e.  $b_\tau(v) = b_\tau(u)$ ), it must be that  $b_\tau(u) = 1$  because there must be a face  $\tau'$  of  $\tau$  of dimension  $\dim(\lambda)$  such that  $u \in \chi(\tau')$ , and by Lemma 11(1),  $b_\tau(\chi(\tau')) = \{1\}$ ; and finally, by definition,  $\text{val}(\lambda) = \{1\}$ .

However, we cannot say that same for any input simplex  $\lambda \in \chi^\ell(\sigma)$  with  $\dim(\lambda) \geq n - 2$ . Consider the case that  $\chi(\lambda)$  does not intersect  $\chi(\tau)$  and the boundary of  $\chi^{\ell+1}(\sigma)$ ; in that case  $b_\tau(\chi(\lambda)) = 1$ , by definition of  $b_\tau$ , but  $\text{val}(\lambda) = \{0, 1\}$ , by definition of  $\text{val}$ . We fix this issue in the second step of the construction: for any  $\lambda \in \chi^\ell(\sigma)$  with  $\dim(\lambda) = n - 2$  and  $b_\tau(\chi(\lambda)) = \{1\}$ , pick the vertex  $v \in \chi(\lambda)$  with smallest id among the vertices with  $\text{carr}(v, \lambda, \chi) = \lambda$  (namely,  $v$  is a vertex with smallest id of the “central”  $(n - 2)$ -simplex of  $\chi(\lambda)$ ), and set  $b_\tau(v) = 0$ .

By construction, we have that  $b_\tau(\chi(\lambda)) = \{0, 1\} = \text{val}(\lambda)$ . Note that  $\lambda$  is not face of  $\tau$  because initially we had  $b_\tau(\chi(\lambda)) = \{1\}$ , which is not true for  $(n - 2)$ -dimensional faces of  $\tau$ , by Lemma 11(2), and if  $\lambda$  intersects  $\tau$ , then  $v \notin \chi(\tau)$  because  $v$  is an “internal” vertex of  $\chi(\lambda)$ . Therefore,  $b_\tau|_{\chi(\tau)}$  remains the same after the second step. Moreover, since we pick vertices with smallest id,  $b_\tau$  remains symmetric. Finally, for any  $\lambda \in \chi^\ell(\sigma)$  with  $\dim(\lambda) = n - 1$ , if  $\lambda = \tau$ , we know already that  $b_\tau(\chi(\lambda)) = \{0, 1\}$ , and if  $\lambda \neq \tau$ , we already saw that for every  $(n - 2)$ -face  $\tau'$  of  $\lambda$ ,  $b_\tau(\chi(\tau')) = \{0, 1\}$ , and thus  $b_\tau(\chi(\lambda)) = \{0, 1\}$ . Therefore, we conclude that  $b_\tau(\chi(\lambda)) = \text{val}(\lambda)$ , for every input simplex  $\tau \in \chi^\ell(\sigma)$ .

Thus, we have shown that  $b_\tau$  is a symmetric binary coloring that is consistent and complete w.r.t.  $\text{val}$ . Also, by Lemma 11,  $b_\tau(\chi(\tau))$  has no monochromatic simplexes of dimension  $n - 1$ . Therefore,  $\langle \chi^\ell(\sigma), \{0, 1\}, \text{val} \rangle$  is locally solvable in one round. ◀

## 5 Local Valency Impossibility Proofs

Here we make precise our notion of “impossibility proof in the FLP style,” and use Theorems 8 and 10 to argue that such impossibility proofs do not exist for  $(n - 1)$ -set agreement and weak symmetry breaking in the IIS model.

In a *local valency impossibility proof* for say, set agreement, one assumes by way of contradiction a hypothetical  $R$ -round protocol solving the task. Recall that the protocol determines valencies, for all simplexes in all rounds, starting with those of the initial configuration  $\sigma$ . The valencies must respect the task specification, since we assume the protocol solves the task. For example,  $\text{val}(P_i, i) = \{i\}$ , where  $(p_i, i) \in \sigma$  is the initial state of  $P_i$  (in an execution where  $P_i$  sees only itself, it must decide its own input value). A crucial observation is that what we are given in a local valency impossibility proof are only the valencies, and there are many protocols that could produce the same valencies (i.e., many simplicial maps  $c$  assigning decisions to  $\chi^R(\sigma)$ , yielding the same valencies).

The proof consists of  $R - 1$  phases to select a sequence of simplexes  $\sigma_0, \sigma_1, \dots, \sigma_{R-1}$ , starting with  $\sigma_0 = \sigma$  and such that  $\sigma_\ell \in \chi^\ell(\sigma)$  for all  $1 \leq \ell \leq R - 1$ , extending the sequence by one at each phase.

Assume we have selected the sequence  $\sigma_0, \dots, \sigma_\ell$ , for some  $\ell \geq 1$ . To select  $\sigma_{\ell+1} \in \chi(\sigma_\ell) \subset \chi^{\ell+1}(\sigma)$ , one considers all simplexes in  $\sigma' \in \chi(\sigma_\ell)$ , together with their valencies,  $\text{val}(\sigma')$ . When we reach phase  $R - 1$ , and we have selected  $\sigma_{R-1} \in \chi(\sigma_{R-2})$ , the protocol reveals all decisions in  $\chi(\sigma_{R-1}) \subset \chi^R(\sigma)$  (and *only* those decisions). Namely, a simplicial map  $c$  assigning a decision to each vertex of  $\chi(\sigma_{R-1})$ , respecting all previously observed valencies, namely, all those in each  $\chi(\sigma_\ell)$ .

There is a *local valency impossibility proof* for the task if and only if one can select a sequence  $\sigma_0, \sigma_1, \dots, \sigma_{R-1}$  such that the task is *not* locally solvable in one round at  $\sigma_{R-1}$ . Namely, if there is no decision function  $c$ , that respects the valencies and is consistent with the task specification. In the case of set agreement, at least one simplex must have  $n$  different decisions, for any  $c$  that respects the valencies.

## 18:14 Locally Solvable Tasks

Therefore, there is no such a proof if we are able to exhibit valencies of a hypothetical protocol such that, for any selection  $\sigma_0, \sigma_1, \dots, \sigma_{R-1}$ , there is a decision function  $c$  corresponding to those valencies that locally solves  $(n-1)$ -set agreement (the argument for weak symmetry breaking is analogous):

- Fix any  $R \geq 2$ .
- For the input complex, the valency of each  $\sigma' \subseteq \sigma$  is  $val(\sigma') = \sigma'$ .
- In phase  $\ell \in \{0, \dots, R-2\}$ , the valency of each simplex  $\tau \in \chi(\sigma_\ell) \subset \chi^{\ell+1}(\sigma)$  is the valency of the simplex in the valency task  $\mathcal{T}^{\ell+1} = \langle \chi^{\ell+1}(\sigma), \sigma, val \rangle$  in Definition 3, namely,  $val(\tau)$ .
- In phase  $R-1$ , the protocol picks a decision map  $c : \chi^R(\sigma) \rightarrow \sigma$  that is consistent and complete w.r.t  $\mathcal{T}^{R-1} = \langle \chi^{R-1}(\sigma), \sigma, val \rangle$  and does not have fully colored  $(n-1)$ -simplexes in  $\chi(\sigma_{R-1}) \subset \chi^R(\sigma)$ , and provides only the decisions  $c(\chi(\sigma_{R-1}))$ . Such a mapping exists since  $\mathcal{T}^{R-1}$  is  $(n-1)$ -locally solvable, due to Theorem 8.

Notice that no matter the simplex  $\sigma_\ell$  we chose in each phase, we cannot find a contradiction in the decisions of  $\chi(\sigma_{R-1})$ . The only thing that remains to be argued is that the valencies are consistent during all phases. More specifically, valencies preserve containment in the same phase and can only shrink as the phases go by, and additionally they do not contradict validity, i.e., the valency of a simplex is a subset of its carrier. Thus, for any  $R$ , there are valencies that could be produced by a hypothetical set agreement protocol. This is implied by the three properties below that are satisfied for every valency task  $\mathcal{T}^\ell = \langle \chi^\ell(\sigma), \sigma, val \rangle$ ,  $\ell \in \{1, \dots, R-1\}$ , and whose proof is based on Observation 12. These properties also show that the decisions of  $\chi^{R-1}(\sigma_{R-1})$  revealed by the protocol are consistent with all valencies in all phases.

► **Observation 12.** For  $\ell \geq 0$ , for every  $\gamma \in \chi^\ell(\sigma)$ ,  $ID(\gamma) \subseteq carr(\gamma, \sigma, \chi^\ell)$ . Furthermore, if  $dim(carr(\gamma, \sigma, \chi^\ell)) = dim(\gamma)$ , then  $carr(\gamma, \sigma, \chi^\ell) = ID(\gamma)$ .

**Containment** For  $\tau, \tau' \in \chi^\ell(\sigma)$  with  $\tau' \subset \tau$ , we have  $val(\tau') \subseteq val(\tau)$ . By Observation 12, for every  $\gamma \in \chi^\ell(\sigma)$ ,  $ID(\gamma) \subseteq carr(\gamma, \sigma, \chi^\ell)$ . Since  $\tau' \subset \tau$ , we have  $carr(\tau', \sigma, \chi^\ell) \subset carr(\tau, \sigma, \chi^\ell)$ . Depending on the dimension of  $\tau$ ,  $val(\tau)$  is either  $ID(\tau)$  or  $carr(\tau, \sigma, \chi^\ell)$ ; and similarly for  $\tau'$ . Therefore,  $val(\tau') \subseteq val(\tau)$ .

**Valencies shrink** Consider any  $m > \ell$  with  $0 \leq \ell + m \leq R-1$  and the valency task  $\mathcal{T}^{\ell+m} = \langle \chi^{\ell+m}(\sigma), \sigma, val' \rangle$ . For  $\tau \in \chi^\ell(\sigma)$  and  $\tau' \in \chi^m(\tau)$ ,  $val'(\tau') \subseteq val(\tau)$ . The argument is very similar to the previous one. Since  $\tau' \in \chi^m(\tau) \subset \chi^\ell(\sigma)$ , we have that  $carr(\tau', \sigma, \chi^{\ell+m}) \subseteq carr(\tau, \sigma, \chi^\ell)$ . Depending on the dimension of  $\tau$ ,  $val(\tau)$  is either  $ID(\tau)$  or  $carr(\tau, \sigma, \chi^\ell)$ , and we have that  $ID(\tau) \subseteq carr(\tau, \sigma, \chi^\ell)$ , by Observation 12; and similarly for  $\tau'$ . Therefore,  $val'(\tau') \subseteq val(\tau)$ .

**Validity** For  $\tau \in \chi^\ell(\sigma)$ , we have  $val(\tau) \subseteq carr(\tau, \sigma, \chi^\ell)$ . If  $dim(\tau)$  is  $n-2$  or  $n-1$ ,  $val(\tau) = carr(\tau, \sigma, \chi^\ell)$ , and if  $dim(\tau) \leq n-3$ ,  $val(\tau) = ID(\tau) \subseteq carr(\tau, \sigma, \chi^\ell)$ , where the last containment follows from Observation 12.

The previous properties hold for all simplexes of the valency tasks in Definition 3, and thus we conclude that there is no contradiction on the valencies provided during the phases.

## 6 Discussion

This paper argues that the  $(n-1)$ -set agreement and weak symmetry breaking (and hence  $(2n-2)$ -renaming) impossibilities in the wait-free read/write shared memory model cannot be proved using local arguments, in the style of FLP. We introduced the notions of *valency task* and *local solvability* for set agreement and weak symmetry breaking. We formalized

the notion of *local-valency impossibility proof* for these tasks, where a presumptive protocol for these tasks can always hide erroneous results, even after committing to valencies one round before termination. We showed that there are no local-valency impossibility proofs for  $(n - 1)$ -set agreement and weak symmetry breaking in the wait-free read/write shared memory model.

Alistarh, Aspnes, Ellen, Gelashvili and Zhu [2] studied a similar question by defining a game between a *prover* and a *protocol*, as a way to represent extension-based techniques for proving impossibility results. They have shown that, for set agreement, a protocol can win this game against any prover, thus showing extension-based techniques do not suffice for proving the impossibility of solving set agreement. Their approach is restricted to *unbounded* protocols. This also complicates the argument, since they need to work with *non-uniform* simplicial subdivisions. In contrast, we consider *bounded wait-free*. This allows to assume that all processes decide at the same round,  $R$  (hence giving more information and power to the prover), leading to simpler *uniform* subdivisions. We stress that there is no loss of generality in this assumption, since a task is wait-free solvable if and only if it is wait-free solvable by a protocol where all processes decide at the same round. Furthermore, while Alistarh et al. study only  $k$ -set agreement, we also investigate weak symmetry breaking, and by reduction, renaming.

Looking forward, it would be interesting to define a notion of valency task that can be applied to any task, such as approximate agreement. Also, in the context of randomized and non-deterministic protocols. Another interesting question is *how much* of the final decisions the protocol can reveal; for example, revealing consistent decisions even if several configurations are chosen instead of only one. Finally, we would like to explore local-valency proofs beyond our wait-free setting, in models that are not round-based [28] or non-compact [30], like  $t$ -resilient models.

---

## References

- 1 M. K. Aguilera and S. Toueg. A simple bivalency proof that  $t$ -resilient consensus requires  $t + 1$  round. *Information Processing Letters*, 71(3-4):155–158, 1999.
- 2 Dan Alistarh, James Aspnes, Faith Ellen, Rati Gelashvili, and Leqi Zhu. Why extension-based proofs fail. In *STOC*, pages 986–996, 2019.
- 3 James Aspnes. Lower bounds for distributed coin-flipping and randomized consensus. *Journal of the ACM*, 45(3):415–450, 1998.
- 4 H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment. *Journal of the ACM*, 37(3):524–548, 1990.
- 5 Hagit Attiya and Armando Castañeda. A non-topological proof for the impossibility of  $k$ -set agreement. *Theoretical Computer Science*, 512:41–48, 2013.
- 6 Hagit Attiya, Armando Castañeda, Danny Hendler, and Matthieu Perrin. Separating lock-freedom from wait-freedom. In *PODC*, pages 41–50, 2018. doi:10.1145/3212734.3212739.
- 7 Hagit Attiya and Keren Censor. Tight bounds for asynchronous randomized consensus. *Journal of the ACM*, 55(5):1–26, 2008.
- 8 Hagit Attiya, Nancy Lynch, and Nir Shavit. Are wait-free algorithms fast? *Journal of the ACM*, 41(1):725–763, 1994.
- 9 Hagit Attiya and Ami Paz. Counting-based impossibility proofs for set agreement and renaming. *Journal of Parallel and Distributed Computing*, 87:1–12, 2016.
- 10 E. Borowsky and E. Gafni. Generalized FLP impossibility result for  $t$ -resilient asynchronous computations. In *ACM Symposium on Theory of Computing*, pages 91–100, 1993.
- 11 Elizabeth Borowsky and Eli Gafni. Immediate atomic snapshots and fast renaming. In *PODC*, pages 41–51, 1993.

- 12 Armando Castañeda and Sergio Rajsbaum. New combinatorial topology bounds for renaming: The lower bound. *Distributed Computing*, 22(5-6):287–301, 2010. doi:10.1007/s00446-010-0108-2.
- 13 Armando Castañeda and Sergio Rajsbaum. New combinatorial topology bounds for renaming: The upper bound. *J. ACM*, 59(1):3:1–3:49, 2012. doi:10.1145/2108242.2108245.
- 14 Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. Unifying concurrent objects and distributed tasks: Interval-linearizability. *J. ACM*, 65(6):45:1–45:42, 2018. doi:10.1145/3266457.
- 15 Soma Chaudhuri. More *choices* allow more *faults*: Set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132–158, 1993.
- 16 Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.
- 17 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.
- 18 Eli Gafni and Sergio Rajsbaum. Distributed programming with tasks. In *OPODIS*, pages 205–218, 2010. doi:10.1007/978-3-642-17653-1\_17.
- 19 Eli Gafni, Sergio Rajsbaum, and Maurice Herlihy. Subconsensus tasks: Renaming is weaker than set agreement. In *DISC*, pages 329–338, 2006. doi:10.1007/11864219\_23.
- 20 M. Herlihy, D. Kozlov, and S. Rajsbaum. *Distributed Computing Through Combinatorial Topology*. Elsevier-Morgan Kaufmann, 2013. doi:10.1016/C2011-0-07032-1.
- 21 Maurice Herlihy. Impossibility results for asynchronous PRAM. In *SPAA*, pages 327–336, 1991. doi:10.1145/113379.113409.
- 22 Maurice Herlihy and Sergio Rajsbaum. The topology of distributed adversaries. *Distrib. Comput.*, 26(3):173–192, June 2013. doi:10.1007/s00446-013-0189-9.
- 23 Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858–923, 1999.
- 24 Maurice P. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):123–149, 1991.
- 25 Idit Keidar and Sergio Rajsbaum. A simple proof of the uniform consensus synchronous lower bound. *Information Processing Letters*, 85(1):47–52, 2003. doi:10.1016/S0020-0190(02)00333-2.
- 26 Wai-Kau Lo and Vassos Hadzilacos. All of us are smarter than any of us: Nondeterministic wait-free hierarchies are not robust. *SIAM J. Comput.*, 30(3):689–728, 2000. doi:10.1137/S0097539798335766.
- 27 M. C. Loui and H. A. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.
- 28 Ronit Lubitch and Shlomo Moran. Closed schedulers: a novel technique for analyzing asynchronous protocols. *Distributed Computing*, 8(4):203–210, 1995. doi:10.1007/BF02242738.
- 29 Yoram Moses and Sergio Rajsbaum. A layered analysis of consensus. *SIAM Journal on Computing*, 31(4):989–1021, 2002.
- 30 Thomas Nowak, Ulrich Schmid, and Kyrill Winkler. Topological characterization of consensus under general message adversaries. In *PODC*, page 218–227, 2019. doi:10.1145/3293611.3331624.
- 31 S. Rajsbaum. Iterated shared memory models. In *LATIN*, pages 407–416, 2010. doi:10.1007/978-3-642-12200-2\_36.
- 32 Michael Saks and Fotios Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. *SIAM Journal on Computing*, 29(5):1449–1483, 2000.
- 33 E. Sperner. Neuer beweis für die invarianz der dimensionszahl und des gebietes. *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg*, 6(1):265–272, 1928. doi:10.1007/BF02940617.



# Approximate Majority with Catalytic Inputs

**Talley Amir**

Yale University, New Haven, CT, USA  
talley.amir@yale.edu

**James Aspnes**

Yale University, New Haven, CT, USA  
james.aspnes@gmail.com

**John Lazarsfeld**

Yale University, New Haven, CT, USA  
john.lazarsfeld@yale.edu

---

## Abstract

Population protocols [6] are a class of algorithms for modeling distributed computation in networks of finite-state agents communicating through pairwise interactions. Their suitability for analyzing numerous chemical processes has motivated the adaptation of the original population protocol framework to better model these chemical systems. In this paper, we further the study of two such adaptations in the context of solving approximate majority: persistent-state agents (or *catalysts*) and spontaneous state changes (or *leaks*).

Based on models considered in recent protocols for populations with persistent-state agents [3, 5, 14], we assume a population with  $n$  catalytic input agents and  $m$  worker agents, and the goal of the worker agents is to compute some predicate over the states of the catalytic inputs. We call this model the Catalytic Input (CI) model. For  $m = \Theta(n)$ , we show that computing the parity of the input population with high probability requires at least  $\Omega(n^2)$  total interactions, demonstrating a strong separation between the CI model and the standard population protocol model. On the other hand, we show that the simple third-state dynamics [7, 20] for approximate majority in the standard model can be naturally adapted to the CI model: we present such a constant-state protocol for the CI model that solves approximate majority in  $O(n \log n)$  total steps with high probability when the input margin is  $\Omega(\sqrt{n \log n})$ .

We then show the robustness of third-state dynamics protocols to the transient leaks events introduced by [3, 5]. In both the original and CI models, these protocols successfully compute approximate majority with high probability in the presence of leaks occurring at each step with probability  $\beta \leq O(\sqrt{n \log n}/n)$ . The resilience of these dynamics to leaks exhibits similarities to previous work involving Byzantine agents, and we define and prove a notion of equivalence between the two.

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

**Keywords and phrases** population protocols, approximate majority, catalysts, leaks, lower bound

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2020.19

**Related Version** A full version of the paper is available at <https://arxiv.org/abs/2009.08847>.

**Funding** *James Aspnes*: Supported in part by NSF grant CCF-1650596.

**Acknowledgements** The authors would like to thank Anne Condon, Monir Hajiaghayi, David Kirkpatrick, and Ján Maňuch for a helpful discussion regarding the analysis of the DBAM protocol. The authors are also grateful for a discussion with Francesco d’Amore, Andrea Clementi, and Emanuele Natale, who pointed out the connection between catalytic agents in population protocols and stubborn agents in other types of multi-agent systems. We also thank the anonymous reviewers for their helpful feedback.



© Talley Amir, James Aspnes, and John Lazarsfeld;  
licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 19; pp. 19:1–19:16

Leibniz International Proceedings in Informatics



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



## 1 Introduction

The population protocol model [6] is a theoretical framework for analyzing distributed computation in ad hoc networks of anonymous, mobile agents: at each step, a random pair of agents is chosen to interact, and their local states are updated according to a global transition function. Population protocols can solve numerous problems in distributed computing, including majority (which is also referred to as consensus) [5, 7, 12], source detection [3, 14], and leader election [4, 16, 17].

Population protocols are a special case of chemical reaction networks (CRNs), which are systems of transition rules describing how a set of chemical reactants stochastically transform into a set of products. In particular, population protocols are chemical reaction networks with exactly two reactants which form two products, where each transition rule for a pair of reactants is weighted with probability 1. Given their suitability for modeling chemical processes, population protocols have been used to study computation not only by chemical reaction networks [10], but also DNA strand displacement [11, 21] and biochemical networks [9]. These applications of population protocols in chemistry have inspired various adaptations of the model. In this paper, we focus on two such variations on population protocols in the context of solving **majority**, the problem of determining which of two states is initially more prevalent in a population.

The first modification to the model we consider, which was introduced to the literature in previous works studying source detection and bit-broadcast [3, 14] and later studied in the context of the majority problem [5, 13], is the presence of **persistent-state agents**, or agents whose state never changes. While some works use persistent-state agents to model authoritative sources of information [14] or “stubborn” nodes that are unwilling to change state [13], others describe these entities as an embodiment of chemical catalysts because they induce a state transition in another agent without themselves changing state [3, 5]. Using the latter perspective, we refer to these persistent-state agents as **catalysts**.

In this work, we call the class of population protocols with catalysts the **catalytic input (CI) model**. We formally define the model to consist of  $n$  **catalytic input agents**, which in accordance with their name do not ever change state, and  $m$  **worker agents** that can change state and wish to compute some function on the states of the catalysts. While the CI model is similar to the standard population protocol model, we show that there exists a strong separation between the two in terms of their computational power.

The next variation on the model we consider is the introduction of transient leak events, studied previously in the contexts of solving source detection and comparison [3, 5]. In brief, a “leak” simulates the low-probability event that a molecule undergoes a reaction that would typically take place in the presence of a catalyst. In population protocols, this is modeled by a spontaneous change of state at a single agent, and note that catalytic agents in the CI model are not susceptible to leaks because they never change state. A leak replaces an interaction between two agents at any given step with some fixed probability, known as the **leak rate** [3]. Although leaks have typically been studied in the presence of catalysts, we consider leaks to more generally model unpredictable or adversarial behavior which may occur in the absence of catalysts as well.

We explore the impact of leaks on third-state dynamics [7, 20] solving majority. Our work demonstrates that third-state dynamics can solve **approximate majority**, or majority with a lower-bounded initial difference between the counts of the two input states, with upper-bounded leak rate both in the standard and CI population models.

## 1.1 Related Work

The third-state dynamics protocol in the original population model (sometimes called *undecided-state dynamics*) was introduced by Anlguin et al. [7] and independently by Perron et al. [20]. An agent is either in a state  $X$  or  $Y$ , or in a *blank* state  $B$  (sometimes called an *undecided* state). The transition rules are shown in Figure 1, and we refer to this protocol as DBAM<sup>1</sup>. Assuming an initial  $X$  majority, a simplified analysis from Condon et al. [12] showed that all  $n$  agents in the population transition to the  $X$  state within  $O(n \log n)$  total interactions with high probability, so long as the input margin  $|X| - |Y|$  at the start of the protocol is at least  $\Omega(\sqrt{n \log n})$ . The DBAM protocol is also robust to a small subset of faulty *Byzantine* agents [7, 12], meaning that all but a  $O(\sqrt{n \log n}/n)$  fraction of the population still reaches the  $X$  state within  $O(n \log n)$  interactions with high probability, despite the presence of these dishonest agents.

The DBAM protocol and similar variants of third-state dynamics have been shown to more generally compute consensus (where all agents converge to either  $X$  or  $Y$ , but where this need not be the initial majority value), both in the original population protocols model [7, 12] and in other similar distributed models [8, 13]. In particular, the closely related results of d’Amore et al. [13] analyzed an analogous version of the DBAM protocol in the synchronous PULL model. The authors considered systems with *stubborn* agents (as in [24]) which are similar to the persistent-state catalytic agents we consider in the present work. However, the parallel synchronous scheduling model considered in [13] is fundamentally distinct from the sequential pairwise scheduling used in population protocols.

The notion of a persistent *source* state in population protocols originated from [14], where sources are used to solve **detection** (the detection of a source in the population) and **bit broadcast** (the broadcast of a 0 or 1 message from a set of source agents). An accompanying work [3] introduces the concept of leaks, or spontaneous state changes, and investigates the detection problem in their presence. Generally, leaks can be dealt with using error-correcting codes [22]; however, for certain problems there are more efficient specialized solutions. For example, Alistarh et al. [3] demonstrate that detection in the presence of leaks (up to rate  $\beta = O(1/n)$ ) can be solved with high probability using  $\log \frac{n}{k} + O(\log \log n)$  states, where  $k \leq n$  is the number of sources in the population.

More recently, [5] examines leaks in the context of the **comparison** problem. Comparison is a generalization of the majority problem, where some possibly small subset of the population is in input state  $X_0$  or  $Y_0$  and the task of the population is to determine which of the two states is more prevalent. Alistarh et al. [5] solve comparison in  $O(n \log n)$  interactions with high probability using  $O(\log n)$  states per agent, assuming  $|X_0| \geq C|Y_0|$  for some constant  $C$ , and  $X_0, Y_0 \geq \Omega(\log n)$ . The protocol is self-stabilizing, meaning that it dynamically responds to changes in the counts of input states.

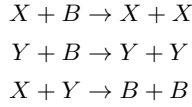
## 1.2 Our Contribution

In this work, motivated by the recent interest in population models with catalytic agents and with transient leaks, we study the well-known third-state dynamics protocols [7, 20] for solving approximate majority in the presence of each of these variants separately as well

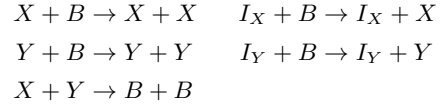
---

<sup>1</sup> DBAM stands for *double-B approximate majority* where *double-B* captures the fact that following an  $X + Y$  interaction, both agents transition to the  $B$  state. This protocol is the two-way variant of the original protocol from [7], which uses one-way communication and where only one agent updates its state per pairwise interaction.

## 19:4 Approximate Majority with Catalytic Inputs



■ **Figure 1** Transition rules for the DBAM protocol [7] in the original population model.



■ **Figure 2** Transition rules for our DBAM-C protocol in the CI model.

as together. To begin, we formalize the CI model consisting of  $n$  catalyts and  $m$  workers, where  $N = n + m$ . While conceptually similar to other models considering these types of catalytic agents [3, 5, 13], introducing the distinction between the two (possibly unrelated) population sizes provides a new level of generality for designing and analyzing protocols in this setting, both with and without leaks.

Although the CI and original population models are almost identical, we show a strong separation between the computational power of the two. When  $m = \Theta(n)$ , we prove a lower bound showing that **parity**, the problem of determining whether the number of catalyts is odd or even, cannot be computed in fewer than  $\Omega(n^2)$  interactions with high probability in the CI model. On the other hand, the result of [19] shows that this predicate *is* computable within  $O(n \text{ polylog } n)$  total steps in the standard model with high probability<sup>2</sup>.

While some problems have strictly different lower bounds on running time in these two models, others do not and can in fact be solved using nearly identical techniques. In particular, we show that the approximate majority problem can be solved in the CI model by naturally extending the DBAM protocol.

In the approximate majority problem in the CI model, each catalytic input agent holds a persistent value of  $I_X$  or  $I_Y$  and each worker agent holds either an undecided, or blank value  $B$ , or an  $X$  or  $Y$  value corresponding to a belief in an  $I_X$  or  $I_Y$  input majority, respectively. The worker agents seek to correctly determine the larger of  $|I_X|$  and  $|I_Y|$  so long as the input margin  $||I_X| - |I_Y||$  is sufficiently large. By adapting the third-state dynamics process [7], we present a constant-state protocol for approximate majority with catalytic inputs called DBAM-C (see Figure 2). The protocol converges with high probability in  $O(N \log N)$  total steps when the initial input margin is  $\Omega(\sqrt{N \log N})$  and  $m = \Theta(n)$ . We then show that this input margin is optimal in the CI model up to a  $O(\sqrt{\log N})$  factor when  $m = \Theta(n)$ .

Moreover, in the presence of transient leak events, we show that both the third-state dynamics protocol in the original model and our adapted protocol in the CI model exhibit a strong robustness to leaks. When the probability of a leak event is bounded, we show that with high probability both protocols still quickly reach a configuration where nearly all agents share the correct input majority value.

Notice that the approximate majority problem in the CI model is equivalent to the comparison problem considered by [5], so we demonstrate how our protocol compares to the results of this work. We show that our DBAM-C protocol converges correctly within the same time complexity of  $O(n \log n)$  total steps, while only using *constant* state space (compared to the logarithmic state used by the protocols in their work). Moreover, in populations where  $m = \Theta(n)$ , our protocol tolerates a less restrictive bound on the input margin compared to [5] ( $\Omega(\sqrt{n \log n})$  compared to  $\Omega(n)$ ). In the presence of transient leaks, our protocol also shows

<sup>2</sup> We define “high probability” to mean with probability at least  $1 - n^{-c}$  where  $n$  is the total number of agents and  $c \geq 1$ .

robustness to a higher leak rate of  $\beta \leq O(\sqrt{n \log n}/n)$ . However, unlike [5], our protocol is not self-stabilizing and requires that the number of inputs be at least a constant fraction of the total population for our main results. In order to achieve these results, we leverage the random walk analysis techniques and analysis structure introduced by [12].

Finally, we compare the impact of leaks on population protocols with that of faulty Byzantine processes. While the fast robust approximate majority protocol of [7] is proven to be robust to a number of Byzantine agents that is bounded by the input margin [7, 12], we show that DBAM is robust to a similarly bounded leak rate and has sampling error matching the result from [12].

The structure of the remainder of the paper is as follows: in Section 2 we introduce notation and definitions central to our results. Section 3 presents our lower bounds over the CI model, which demonstrates the separation between the CI and original population models. In Section 4, we analyze the correctness and efficiency of the DBAM-C protocol for approximate majority in the CI model, and in Section 5 we demonstrate the leak-robustness of both the DBAM-C and original DBAM protocols. Then in Section 6, we compare the notion of transient leaks with the adversarial Byzantine model, demonstrating parallels between previous results examining Byzantine behavior and our work.

Throughout the paper, we provide overviews of the intuition and techniques used to obtain our results and defer most proofs to the full version.

## 2 Preliminaries

We begin with some definitions. Denote by  $N$  the number of agents in the population.

### Population Protocols

Population protocols are a class of algorithms which model interactions between mobile agents with limited communication range. Agents only interact with one another if they are within close enough proximity of each other. In order to model this type of system in an asynchronous setting, interactions between pairs of agents are executed in sequence. The interaction pattern of these agents is dictated by a **scheduler**, which may be random or adversarial. In this work we will assume that the scheduler is uniformly random, meaning that an ordered pair of agents is chosen to interact at each time step independently and uniformly at random from all  $N(N-1)$  ordered pairs of agents in the system.

As defined by [6] which first introduced the model, a population protocol  $\mathcal{P}$  consists of a **state set**  $\mathcal{S} = \{s_1, s_2, \dots, s_k\}$ , a **rule set**  $\mathcal{R} : \mathcal{S}^2 \mapsto \mathcal{S}^2$ , an **output alphabet**  $\mathcal{O}$ , and an **output function**  $f : \mathcal{S} \mapsto \mathcal{O}$ . The output function computes the evaluation of some function on the population locally at each agent. The **configuration** of the population is denoted as a vector  $\mathbf{c} = \langle c_1, c_2, \dots, c_k \rangle$  such that each  $c_i \geq 0$  is equal to the number of agents in the population in state  $s_i$ , from which it follows that  $\sum c_i = N$ . For convenience, we denote by  $|s_i|$  the number of agents in the population in state  $s_i$ .

At each point in time, the scheduler chooses an ordered pair of agents  $(a_i, a_j)$ , where  $a_i$  is the **initiator** and  $a_j$  is the **responder** [6]. The agents interact and update their state according to the corresponding rule in  $\mathcal{R}$ . In general, a rule in  $\mathcal{R}$  is written as  $A+B \rightarrow C+D$  to convey that two agents, an initiator in state  $A$  and a responder in state  $B$ , interact and update their states to be  $C$  and  $D$ , respectively. By convention,  $N/2$  interactions make one unit of **parallel time** [7]. This convention is equivalent to assuming every agent interacts once per time unit on average.

## 19:6 Approximate Majority with Catalytic Inputs

An **execution** is the sequence of configurations of a run of the protocol, which **converges** when the population arrives at a configuration  $\mathbf{d}$  such that all configurations chronologically after  $\mathbf{d}$  have the same output at each agent as those in  $\mathbf{d}$  [6]. In order to determine the success or failure of an execution of  $\mathcal{P}$ , we will consider a sample of the population to signify the outcome of the protocol [3]. After the expected time to converge, one agent is selected at random and its state is observed. The output associated with the agent's state is considered the output of the protocol. The probability of sampling an agent whose state does not reflect the desired output of the protocol is called the **sample error rate**. Multiple samples can be aggregated to improve the rate of success.

### Catalysts and Leaks

Following [3], in an interaction of the form  $A + B \rightarrow A + D$ , we say  $A$  *catalyzes* the transformation of the agent in state  $B$  to be in state  $D$ . If  $A$  catalyzes every interaction it participates in,  $A$  is referred to as a **catalyst**.

In chemistry, a reaction that occurs in the presence of a catalyst also occurs at a lower rate in the absence of that catalyst. For this reason, recent work in DNA strand displacement, chemical reactions networks, and population protocols [3, 5, 21] have studied the notion of *leakage*: When a catalytic reaction  $A + B \rightarrow A + D$  is possible, then there is some probability that a transition  $B \rightarrow D$  can occur without interacting with  $A$  at all. This type of event, called a **leak**, was introduced in [21].

The probability with which the non-catalyzed variation of a reaction takes place is the **leak rate**, which we denote by  $\beta$ . We simulate a leak as follows: At each step in time, with probability  $1 - \beta$ , the scheduler samples an ordered pair of agents to interact with one another as described in the beginning of the section; the rest of the time (i.e. with probability  $\beta$ ) one agent is chosen uniformly at random from all possible agents and the **leak function**  $\ell : \mathcal{S} \rightarrow \mathcal{S}$  is applied to update this agent's state. Note that we only consider *non-catalytic* agents to be susceptible to these faulty events.

### Catalytic Input Model

In this work, we formalize a **catalytic input** (CI) model consisting of  $n$  catalytic agents that supply the input and  $m$  worker agents that perform the computation and produce output. We define  $N = m + n$  to be the total number of agents in the population. At each time step, the scheduler samples any two agents in the population to interact with one another. If two catalysts are chosen to interact, then the interaction is considered to be **null** as no nontrivial state transition occurs. When  $n = o(m)$ , the probability that two catalysts are chosen to interact is upper bounded by a constant, and so the total running time of the protocol is asymptotically equivalent to the number of non-null interactions needed to reach convergence. In the CI model, we consider convergence to be a term that refers to the states of the worker agents only, as the catalytic agents never change state. Namely, for the approximate majority problem, successful convergence equates to the *worker agents* being in the majority-accepting state. In general, we wish to obtain results that hold with high probability with respect to the *total* number of agents  $N$ .

## 3 Catalytic Input Model Lower Bounds

In this section, we characterize the computational power of the CI population protocol model. Using information-theoretic arguments, we prove two lower bounds over the catalytic model when the number of input agents is a constant fraction of the total population:

► **Theorem 1.** *In the catalytic input model with  $n$  input agents and  $m = \Theta(n)$  worker agents, any protocol that computes the parity of the inputs with probability at least  $1 - N^{-\gamma}$  requires at least  $\Omega(N^2)$  total steps for any  $\gamma \geq 1$ .*

► **Theorem 2.** *In the catalytic input model with  $n$  input agents and  $m = \Theta(n)$  worker agents, any protocol that computes the majority of the inputs within  $O(N \log N)$  total steps requires an input margin of at least  $\Omega(\sqrt{N})$  to be correct with probability at least  $1 - N^{-\gamma}$  for any  $\gamma \geq 1$ .*

The first result can be viewed as a separation between the CI and original population models: since it is shown in [19] that the parity of agents can be computed in the original model within  $O(\text{polylog } n)$  parallel time with high probability, our result indicates that not all semi-linear predicates over the input population in the CI model can be computed in sub-linear parallel time with high probability. Additionally, this rules out the possibility of designing fast protocols for exact majority in the CI model when the input size is a constant fraction of the entire population. On the other hand, the second result indicates the existence of a predicate – approximate majority – that does not require a large increase in convergence time to be computed with high probability in this new model.

One key characteristic of a CI population is the inability for worker agents to distinguish which inputs have previously interacted with a worker. Instead, every worker-input interaction acts like a random sample with replacement from the input population. For proving lower bounds in this model, this characteristic of a CI population leads to the following natural argument: consider a population of  $n$  catalytic input agents and a worker population consisting of a single **super-agent**. Here, we assume the super-agent has unbounded state and computational power, and it is thus able to simulate the entire worker population of any protocol with more workers. In this simulation, any interaction between a worker and an input agent is equivalent to the super-agent interacting with an input chosen uniformly at random: in other words, as a sample with replacement from the input population. Thus we view the super-agent as running a central randomized algorithm to simulate the random interactions that occur in population protocols. If the super-agent needs  $S$  samples to compute some predicate over the inputs with high probability, then so does any multi-worker protocol in the CI model. We denote this information-theoretic model as the **Super CI model**, and restate the above argument more formally in the following lemma.

► **Lemma 3.** *Consider a population with  $n$  catalytic input agents and a worker population consisting of a single super-agent  $W$ . Let  $P$  be a predicate over the input population that requires  $S$  total interactions between  $W$  and the input population in order for  $W$  to correctly compute  $P$  with probability  $\epsilon$ . Then for a CI population with  $n$  catalytic inputs and  $m$  worker agents, computing  $P$  correctly with probability  $\epsilon$  requires at least  $S$  total interactions.*

### Proof Sketch of Theorem 1

In a CI model population with  $n$  input agents and  $m$  worker agents where  $m = \Theta(n)$ , Theorem 1 shows that computing the parity or exact majority of the inputs requires at least  $\Omega(n^2) = \Omega(N^2)$  total interactions to be correct with high probability. We prove this by showing that in the Super CI model described in the previous section, a computationally unbounded super-agent  $W$  requires at least  $\Omega(n^2)$  samples of the input population to correctly compute the input parity with high probability. Applying Lemma 3 then gives Theorem 1.

More formally, for an input population  $C$  of  $n$  agents, each with input value 0 or 1, the parity of  $C$  is said to be 1 if an odd number of agents have input value 1, and 0 otherwise. Now, consider the majority predicate over  $C$ , which is simply the majority value of the input



## 19:8 Approximate Majority with Catalytic Inputs

population. Letting  $X$  denote the number of 1-inputs, and  $Y$  the number of 0-inputs, we refer to the *input margin* of the population  $C$  as the quantity  $|X - Y|$ . Suppose that  $n$  is odd and the input margin of  $C$  is 1. Then  $X$  and  $Y$  are either  $\lfloor \frac{n}{2} \rfloor$  and  $\lceil \frac{n}{2} \rceil$  or vice versa. These two cases can be distinguished either by computing the majority predicate or the parity predicate, making both of these problems equivalent to distinguishing the two cases under this constraint on the input. We will now argue that distinguishing these cases in the Super CI model requires  $\Omega(n^2)$  samples.

Recall that in the Super CI model, a predicate over the input population  $C$  is computed by a single super agent worker  $W$  with unbounded computational power. Thus, the output of  $W$  can be viewed as a mapping between a string of input values obtained from interactions with between  $W$  and the input population and the output set  $\{0, 1\}$ . We refer to interactions between  $W$  and the input population as **samples** of the input, and for a fixed number of samples  $S$ , we refer to  $W$ 's output as its **strategy**.

First, we show that for some fixed distribution over the input values of  $C$ , the strategy that maximizes  $W$ 's probability of correctly outputting the majority value of  $C$  is simply to output the majority value of its samples. Let  $I \in \{0, 1\}^S$  be the **sample string** representing the  $S$  independent samples with replacement taken by  $W$ , and let  $\mathbb{S}$  denote the set of all  $2^S$  possible sample strings. We model the population of input agents as being generated by an adversary. Specifically, let  $M$  denote the majority value (0 or 1) of the input population, where we treat  $M$  as a random variable whose distribution is unknown. In any realization of  $M$ , we assume a fixed fraction  $p > 1/2$  of the inputs hold the majority value. Given an input population, the objective of the worker agent is to correctly determine the value of  $M$  through its input sample string  $I$ . By Yao's principle [23], the error of any randomized algorithm (i.e., the randomized simulation run by the super-agent) on the worst case value of  $M$  is no smaller than the error of the best deterministic algorithm on some fixed distribution over  $M$ . So our strategy is to pick a distribution over  $M$ , and to use the the error of the best deterministic strategy with respect to this distribution as a lower bound on the worst-case error of any randomized algorithm used by the super-agent.

Thus, assuming  $M$  is chosen according to some fixed distribution, we model the worker's strategy as a fixed map  $f : \{0, 1\}^S \rightarrow \{0, 1\}$ . Letting  $\mathcal{F}_S$  denote the set of all such maps,  $W$  then faces the following optimization problem:  $\max_{f \in \mathcal{F}_S} \Pr[f(I) = M]$ . For a given  $f \in \mathcal{F}_S$ , let  $p_f = \Pr[f(I) = M]$ , and let  $\Phi \in \mathcal{F}_S$  denote the map that outputs the majority value of the input sample string  $I$ . In the following lemma, we show that when the distribution over  $M$  is uniform, setting  $f := \Phi$  maximizes  $p_f$ . In other words, to maximize the probability of correctly guessing the input population majority value, the worker's optimal strategy is to simply guess the majority value of its  $S$  independent samples. The proof of the lemma simply uses the definitions of conditional probability and the Law of Total Probability to obtain the result.

► **Lemma 4.** *Let  $I = \{0, 1\}^S$  be a sample string of size  $S$  drawn from an input population with majority value  $M$  and majority ratio  $p$ , and assume  $\Pr[M = 1] = \Pr[M = 0] = 1/2$ . Then  $\Pr[\Phi(I) = M] \geq \Pr[f(I) = M]$  for all maps  $f \in \mathcal{F}_S$ , where  $\Phi$  is the map that outputs the majority value of the sample string  $I$ .*

We have established by Lemma 4 that to correctly output the input population majority, the super worker agent's error-minimizing strategy is to output the majority of its  $S$  samples when the distribution over  $M$  is uniform. Now the following lemma shows that when the input margin of the population is 1, this strategy requires at least  $\Omega(n^2)$  samples in order to output the input majority with probability at least  $1 - n^{-c}$  for some constant  $c \geq 1$ . The proof uses a tail bound on the Binomial distribution to show the desired trade off between the error of probability and the requisite number of samples needed to achieve this error.



► **Lemma 5.** *Let  $C$  be a Super CI population of  $n$  agents with majority value  $M$  and input margin 1, and consider an input sample string  $I = \{0, 1\}^S$  obtained by a super worker agent  $W$ . Then for any  $c \geq 1$ , letting  $\Phi(I)$  denote the sample majority of  $I$ ,  $\Pr[\Phi(I) \neq M] \leq n^{-c}$  only holds when  $S \geq \Omega(n^2)$ .*

The proof of Theorem 1 follows from Lemmas 3, 4, and 5 by invoking Yao’s principle.

## Overview of Theorem 2

As mentioned, Theorem 1 implies a strong separation between the CI model and original population model, as [19] and [1, 2] have shown that both parity and majority are computable with high probability within  $O(n \text{ polylog } n)$  total steps in the original model, respectively. Thus, the persistent-state nature of input agents in the CI model may seem to pose greater challenges than in the original model for computing predicates quickly with high probability. However, using the same sampling-based lower bound techniques developed in the preceding section, Theorem 2 shows that when  $m = \Theta(n)$ , and when restricted only to  $S = O(n \log n)$  total steps, any protocol computing majority in the CI model requires an input margin of at least  $\Omega(\sqrt{n}) = \Omega(\sqrt{N})$  to be correct with high probability in  $N$ .

Moreover, in Section 4 we present a protocol for approximate majority in the CI model that converges correctly with high probability within  $O(N \log N)$  total steps, so long as the initial input margin is  $\Omega(\sqrt{N \log N})$ . Thus, the existence of such a protocol indicates that the  $\Omega(\sqrt{N})$  lower bound on the input margin is nearly tight (up to  $\sqrt{\log N}$  factors) for protocols limited to  $O(N \log N)$  total steps when  $m = \Theta(n)$ .

## 4 Approximate Majority with Catalytic Inputs

We now present and analyze the DBAM-C protocol for computing approximate majority in the CI model. The protocol is a natural adaptation of a third-state dynamics from the original model, where we now account for the behavior of  $n$  catalytic input agents and  $m$  worker agents. Using the CI model notation introduced in Section 2, we consider a population with  $N = n + m$  total agents. Each input agent begins (and remains) in state  $I_X$  or  $I_Y$ , and we assume each worker agent begins in a *blank* state  $B$ , but may transition to states  $X$  or  $Y$  according to the transition rules found in Figure 1. Letting  $i_X$  and  $i_Y$  (and similarly  $x, y$  and  $b$ ) be random variables denoting the number of agents in states  $I_X$  and  $I_Y$  (and respectively  $X, Y$ , and  $B$ ), we denote the *input margin* of the population by  $\epsilon = |i_X - i_Y|$ . Throughout the section, we assume without loss of generality that  $i_X \geq i_Y$ .

Intuitively, an undecided (blank) worker agent adopts the state of a decided agent (either an input or worker), but decided workers only revert back to a blank state upon interactions with other workers of the opposite opinion. Thus the protocol shares the opinion-spreading behavior of the original DBAM protocol, but note that the inability for decided worker agents to revert back to the blank state upon subsequent interactions with an input allows the protocol to converge to a configuration where all workers share the same  $X$  or  $Y$  opinion.

The main result of the section characterizes the convergence behavior of the DBAM-C protocol when the input margin  $\epsilon$  is sufficiently large. Recall that we say the protocol *correctly computes the majority* of the inputs if we reach a configuration where  $x = m$ . The following theorem shows that, subject to mild constraints on the population sizes, when the input margin is  $\Omega(\sqrt{N \log N})$ , the protocol correctly computes the majority value of the inputs in roughly logarithmic parallel time with high probability.

## 19:10 Approximate Majority with Catalytic Inputs

► **Theorem 6.** *There exists some constant  $\alpha \geq 1$  such that, for a population of  $n$  inputs,  $m$  workers, and initial input margin  $\epsilon \geq \alpha\sqrt{N \log N}$ , the DBAM-C protocol correctly computes the majority value of the inputs within  $O\left(\frac{N^4}{m^3} \log N\right)$  total interactions with probability at least  $1 - N^{-c}$  for any  $c \geq 1$  when  $m \geq n/10$  and  $N$  is sufficiently large.*

Because the CI model allows for distinct (and possibly unrelated) input and worker population sizes, we aim to characterize all error and success probabilities with respect to the total population size  $N$ . The analysis in the proof of Theorem 6 characterizes the convergence behavior of the protocol in terms of both population sizes  $m$  and  $n$ , and thus the convergence time of  $O((N^4/m^3) \log N)$  is not always equivalent to  $O(N \log N)$ . On the other hand, in the case when  $m = \Theta(n)$  – which is an assumption used to provide lower bounds over the CI model from Section 3 – we have as a corollary (further below) that the protocol correctly computes the majority of the inputs within  $O(N \log N)$  total steps with probability at least  $1 - N^{-\alpha}$ .

### Analysis Overview

The proof of the main result leverages and applies the random walk tools from [12] (in their analysis of the original DBAM protocol) to the DBAM-C protocol. Given the uniformly-random behavior of the interaction scheduler, the random variables  $x, y$  and  $b$  (which represent the count of  $X, Y$ , and  $B$  worker agents in the population) each behave according to some one-dimensional random walk, where the biases in the walks change dynamically as the values of these random variables fluctuate. Based on the coupling principle that an upper bound on the number of steps for a random walk with success probability  $p$  to reach a certain position is an *upper bound* on the step requirement for a second random walk with probability  $\hat{p} \geq p$  to reach the same position, we make use of several *progress measures* that give the behavior of the protocol a natural structure. As used in the analysis of Condon et al. [12], we define  $\hat{x} = x + b/2$ ,  $\hat{y} = y + b/2$ , and  $P = \epsilon + \hat{x} - \hat{y}$ . It can be easily seen that  $\hat{x} + \hat{y} = m$  will hold throughout the protocol. On the other hand, the progress measure  $P$  captures the collective gap between the majority and non-majority opinions in the population. Observe that the protocol has correctly computed the input majority value when  $P = \epsilon + m$  and  $\hat{y} = 0$ .

Our analysis uses a structure of **phases** and **stages** to prove the correctness and efficiency of the protocol. Every correctly-completed stage of Phase 1 results in the progress measure  $P$  doubling, and the phase completes correctly once  $P$  is at least  $\epsilon$  plus some large constant fraction of  $m$ . Then, every correctly-completed stage of Phase 2 results in the progress measure  $\hat{y}$  decreasing by a factor of two, and the phase completes correctly once  $\hat{y}$  drops to  $O(\log m)$ . Finally, Phase 3 of the protocol ends correctly once  $\hat{y}$  drops to 0. The details of this structure are stated formally in the full version of the paper.

Note that among the protocol's non-null transitions (see Figure 2), only the interactions  $I_X + B$ ,  $I_Y + B$ ,  $X + B$ , and  $Y + B$  change the value of either progress measure. For this reason, we refer to the set of non-null transitions (which includes  $X + Y$  interactions) as **productive** steps, and the subset of interactions that change our progress measures as the set of **blank-consuming** productive steps. The analysis strategy for every phase and stage is to employ a combination of standard Chernoff bounds and martingale techniques (in general, see [18] and [15]) to obtain with-high-probability estimates of (1) the number of productive steps needed to complete each phase/stage correctly, and (2) the number of total steps needed to obtain the productive step requirements. Given an input margin that is sufficiently large, and also assuming a population where the number of worker agents is at least a small constant fraction of the input size, we can then sum over the error probabilities of each phase/stage and apply a union bound to yield the final result of Theorem 6.

While the DBAM-C protocol is conceptually similar to the original DBAM protocol, the presence of persistent-state catalysts whose opinions never change requires a careful analysis of the convergence behavior. Moreover, simulation results presented in Section 5 show interesting differences in the evolution of the protocol for varying population sizes. As a simple corollary of Theorem 6, we also state the following result, which simplifies the convergence guarantees of the DBAM-C protocol in the case when  $m = \Theta(n)$ .

► **Theorem 7.** *There exists some constant  $\alpha \geq 1$  such that, for a population of  $n$  inputs,  $m = cn$  workers where  $c \geq 1$ , and an initial input margin  $\epsilon \geq \alpha\sqrt{N \log N}$ , the DBAM-C protocol correctly computes the majority of the inputs within  $O(N \log N)$  total interactions with probability at least  $1 - N^{-a}$  for any  $a \geq 1$  when  $N$  is sufficiently large.*

We note that the result of Corollary 7 implies that the CI model input margin lower bound from Theorem 2 is tight up to a multiplicative  $O(\sqrt{\log N})$  factor.

## 5 Approximate Majority with Transient Leaks

We now consider the behavior of the DBAM and DBAM-C protocols in the presence of transient leak faults. Even in the presence of these adversarial events (which occur up to some bounded rate  $\beta$ ), both the DBAM and DBAM-C protocols will, with high probability, reach configurations where nearly all agents share the input majority opinion. In the presence of leaks, we consider the approximate majority predicate to be computed correctly upon reaching these low sample-error configurations.

Recall that a transient leak is an event where an agent spuriously changes its state according to some leak function  $\ell$ . For example, we denote by  $U \rightarrow V$  the event that an agent in state  $U$  transitions to state  $V$  due to a leak event, where the timing of such events are dictated by the random scheduler and occur with probability  $\beta$  at each subsequent interaction step. In both the DBAM and DBAM-C protocols, the only state changes that could possibly take place due to leaks are  $X \rightarrow B$ ,  $Y \rightarrow B$ ,  $B \rightarrow X$ , and  $B \rightarrow Y$  because these describe all possible state changes that could take place in the presence of an interacting partner. However, our analysis considers an *adversarial* leak event  $X \rightarrow Y$ , which maximally decreases our progress measures and can be considered the “worst” possible leak. Though this leak event is not chemically sound (because no normal interaction can cause an  $X$  agent to transition to the  $Y$  state), our results demonstrate that both DBAM and DBAM-C protocols are robust to this strong adversarial leak event. Thus in a more realistic chemically sound setting, our results will also hold, as the set of transitions working against our progress measures are weaker.

### Leak Robustness of the DBAM Protocol

We start by showing the leak-robustness of the DBAM protocol for approximate majority in the original population protocol model. Recall that in the standard model, *all* agents are susceptible to leaks. Our main result shows that when the leak rate  $\beta$  is sufficiently small, the protocol still reaches a configuration with bounded *sample error* (the proportion of agents in the non-initial-majority state) within  $O(n \log n)$  total interactions with high probability. Unlike the scenario without leak events, note that the protocol will never be able to fully converge to a configuration where all agents remain in the majority opinion. However, reaching a configuration where despite leaks, *nearly* all agents hold the input majority value state matches similar results of [3, 5, 7, 12]. Formally, we have the following Theorem, which characterizes the eventual sample error of the protocol with respect to the magnitude of the leak rate  $\beta$ .

## 19:12 Approximate Majority with Catalytic Inputs

► **Theorem 8.** *There exists some constant  $\alpha \geq 1$  such that, for a population with initial input margin  $\epsilon \geq \alpha\sqrt{n \log n}$  and adversarial leak rate  $\beta \leq (\alpha\sqrt{n \log n})/12672n$ , an execution of the DBAM protocol will reach a configuration with*

1. *sample error  $O(\log n/n)$  when  $\beta \leq O(\log n/n)$*
  2. *sample error  $O(\beta)$  when  $\omega(\log n/n) \leq \beta \leq (\alpha\sqrt{n \log n})/12672n$*
- within  $O(n \log n)$  total interactions with probability at least  $1 - n^{-c}$  for any  $c \geq 1$  when  $n$  is sufficiently large.*

To prove Theorem 8, we again make modified use of the random walk tools from [12]. Using the progress measures  $\hat{y} = y + b/2$  and  $P = \hat{x} - \hat{y}$ , observe that an  $X \rightarrow Y$  leak event incurs twice as much negative progress to both measures as opposed to  $X + B$  events. Compared to the analysis from the non-leak setting, the analysis *with* adversarial leaks must account for the stagnation (or potentially the reversal) of the protocol’s progress toward reaching a high-sample-error configuration. Note that since the sample error of a configuration is defined to be  $(y + b)/n$  (since we assume an initial  $x$  majority wlog), we will use the value  $\hat{y}/n$  to approximate a configuration’s sample error.

We use a similar structure of phases and stages as in the previous section, and we list these details formally in the full version of this work. In this leak-prone setting, we refer to *productive interactions* as any of the non-null transitions found in Figure 1 in addition to a leak event. The set of three non-null and non-leak transitions are referred to as *non-leak productive steps*. For each phase and stage, we obtain high-probability estimates on the number of productive and total steps needed to complete the phase/stage correctly in two steps: first, we bound the number of leak events that can occur during a fixed interval of productive events, and we then show that a smaller sub-sequence of non-leak productive steps is sufficient to ensure that enough progress is made to offset the negative progress of the leaks. We again rely on a combination of Chernoff concentration bounds and martingale inequalities in order to show this progress at every phase and stage.

Theorem 8 also separates the behavior of the protocol into two classes: when  $\beta \leq O(\log n/n)$  (*small* leak rate), and when  $\omega(\log n/n) \leq \beta \leq O(\sqrt{n \log n}/n)$  (*large* leak rate). When the leak rate is large, the probability of a leak event conditioned on a productive step becomes roughly equal to the conditional probability of a non-leak productive step when  $\hat{y} = O(\beta n)$ . Thus, we cannot expect the protocol to make further “progress” toward a lower sample-error configuration with high probability beyond  $\hat{y} = O(\beta n)$ . The same holds for small leak rate when  $\beta = O(\log n/n)$ , and for even smaller values of  $\beta$ , our analysis tools only allow for the high-probability guarantee that  $\hat{y}$  eventually drops to  $O(\log n)$ . the protocol reaches a configuration with  $\hat{y} = O(\log n)$ . In the full version of the paper, we give additional arguments showing that the protocol remains in a configuration with sample error  $O(\log n/n)$  for small leak rate, and with sample error  $O(\sqrt{n \log n}/n)$  for large leak rate, for at least a polynomial number of interactions with high probability.

### Leak Robustness of the DBAM-C Protocol

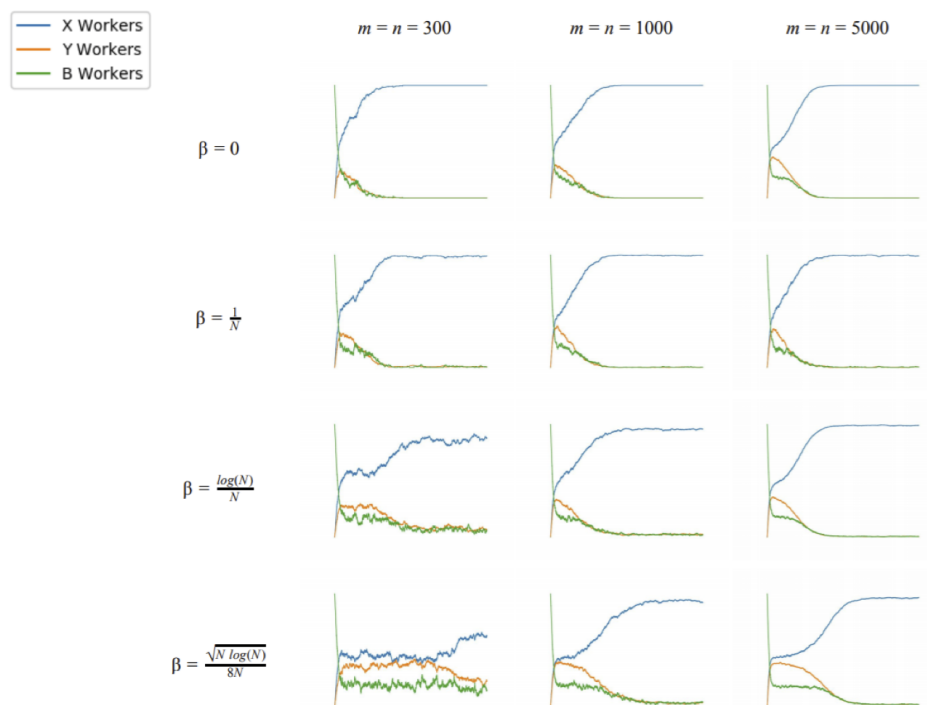
The analysis of the previous subsection is adapted to show that the DBAM-C protocol also exhibits a similar form of leak-robustness in the CI model, and in the following theorem we prove the case where  $m = \Theta(n)$ . Recall that in the CI model, only the non-catalytic worker agents are susceptible to leak events.

► **Theorem 9.** *There exist constants  $\alpha, d \geq 1$  such that, for a population with  $m = cn$  for  $c \geq 1$  and input margin  $\epsilon \geq \alpha\sqrt{N \log N}$ , the DBAM-C protocol will reach a configuration with*

1. *sample error  $O(\log N/N)$  when  $\beta \leq O(\log N/N)$*

2. *sample error  $O(\beta)$  when  $\omega(\log N/N) \leq \beta \leq (\alpha\sqrt{N \log N})/dN$  within  $O(N \log N)$  total interactions with probability at least  $1 - N^{-a}$  for  $a \geq 1$  when  $N$  is sufficiently large.*

The proof of the theorem uses the same progress measures and phase and stage structure introduced in the non-leak setting in Section 4, and the final sample-error guarantee of the protocol is again defined with respect to the magnitude of the leak rate. The behavior of the protocol between the two classes of leak rate is similar as in the DBAM analysis, and the formal details can be found in the full version of the paper. Note that as the upper bound on the leak rate  $\beta$  is a decreasing function in  $N$ , the sample error guarantees of both protocols increase with population size. This relationship is shown across various simulations of the DBAM-C protocol in Figure 3. Moreover, Figure 4a depicts aggregate sample data over many executions of the DBAM-C protocol for varying values of  $N$ , and Figure 4b illustrates the logarithmic parallel time needed to reach convergence in the non-leak setting.

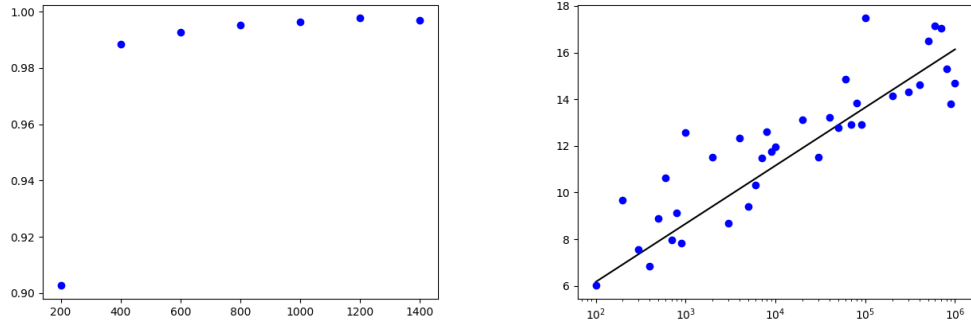


■ **Figure 3** Simulations of the DBAM-C protocol, where each subplot shows the proportion of  $X$ ,  $Y$  and  $B$  worker agents evolve over the course of an execution. All simulations are for  $m = n$ , input margin  $\epsilon = \sqrt{N \log N}$  (with an  $I_X$  majority), and varying values of leak rate  $\beta$  over  $4N \log N$  total interactions. Note that these plots are of single executions and thus provide a qualitative illustration of behavior, rather than statistically significant data. However, we can see that for larger values of  $m = n$  and smaller values of  $\beta$ , the number of  $X$  worker agents reaches a larger count more quickly.

## 6 Leaks Versus Byzantine agents

The original third-state dynamics approximate majority protocol [7] is robust to a bounded number of Byzantine agents, and as shown in the previous sections, both the DBAM protocol and the DBAM-C protocol in the CI model are robust to a bounded leak rate. In this section,

## 19:14 Approximate Majority with Catalytic Inputs



(a) Sample success rate ( $y$ -axis) averaged over 3000 executions of DBAM-C for  $n = 600$ ,  $\Delta_0 = \sqrt{N \log N}$ , and  $\beta = 1/N$ , and varying values of  $m$  ( $x$ -axis). Samples were drawn uniformly from the worker population after  $4N \log N$  total interactions. (b) Parallel time ( $y$ -axis) for DBAM-C without leaks to reach consensus for varying population sizes ( $x$ -axis) where  $m = 2n$ . Data points represent a single execution. The solid black line is  $\frac{3}{4} \log_2(N)$ , showing that convergence takes  $O(N \log N)$  interactions.

■ **Figure 4** Success rate and running time of DBAM-C over various executions of the protocol.

we consider the connection between these two types of faulty behavior. While leaks can occur at any agent with fixed probability throughout an execution, Byzantine agents are a fixed subset of the population, and while a leak event does not change the subsequent behavior of an agent, Byzantine agents may continue to misbehave forever. However, there are parallels between these two models of adversarial behavior. A leak at one agent can cause additional agents to deviate from a convergent configuration; similarly, interactions among non-Byzantine agents, some of which have deviated from a convergent configuration by interacting with a Byzantine agent, can cause additional non-Byzantine agents to diverge.

We prove that for the DBAM and DBAM-C protocols, introducing a leak rate of  $\beta$  has the same asymptotic effect as introducing  $O(\beta N)$  Byzantine agents to the population, which demonstrates an equivalence between these two notions of adversarial behavior among the class of third-state dynamics protocols. Although the results of the previous section assumed leaks that do not follow the laws of chemistry, the following result considers **weak leaks**, which cause the selected agent to decrease its confidence in the majority value by one degree (i.e. a leak causes an agent in state  $X$  to transition to  $B$  and an agent in state  $B$  to transition to  $Y$ , matching the  $X + Y$  and  $Y + B$  transitions). For our purposes, we define two adversarial models  $\mathcal{M}_1$  and  $\mathcal{M}_2$  to be **equivalent** for some protocol  $\mathcal{P}$  if  $\mathcal{P}$  converges to the same asymptotic sample error rate in the same asymptotic running time in both models. We then have the following equivalence result:

► **Theorem 10.** *A population of  $N$  agents running DBAM (or DBAM-C) with weak leak rate  $O(\beta)$  is equivalent to a population of  $N + B$  agents, where  $B = O(N\beta)$  agents are Byzantine, running DBAM (or DBAM-C) without leaks, where in either setting the protocol converges in  $O(N \log N)$  interactions with error probability  $O(\beta)$ .*

## 7 Conclusion and Open Problems

We have shown that third-state dynamics can be used to solve approximate majority with high probability in  $O(n \log n)$  steps up to leak rate  $\beta = O(\sqrt{n \log n}/n)$ , both in the standard population protocol model as well as the CI model when  $m = \Theta(n)$ . While we showed a separation between the CI and original population models, it remains an open question what



other problems (similar to approximate majority) can be computed quickly in the CI model. Additionally, identifying which families of protocols are naturally robust to leak events in the original population model (similar to third-state dynamics) also remains an open question.

---

## References

- 1 Dan Alistarh, James Aspnes, David Eisenstat, Rati Gelashvili, and Ronald L. Rivest. Time-space trade-offs in population protocols. In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 2560–2579. SIAM, 2017. doi:10.1137/1.9781611974782.169.
- 2 Dan Alistarh, James Aspnes, and Rati Gelashvili. Space-optimal majority in population protocols. In Artur Czumaj, editor, *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 2221–2239. SIAM, 2018. doi:10.1137/1.9781611975031.144.
- 3 Dan Alistarh, Bartłomiej Dudek, Adrian Kosowski, David Soloveichik, and Przemysław Uznański. Robust detection in leak-prone population protocols. In *International Conference on DNA-Based Computers*, pages 155–171. Springer, 2017.
- 4 Dan Alistarh and Rati Gelashvili. Polylogarithmic-time leader election in population protocols. In *Proceedings, Part II, of the 42Nd International Colloquium on Automata, Languages, and Programming - Volume 9135, ICALP 2015*, pages 479–491, Berlin, Heidelberg, 2015. Springer-Verlag. doi:10.1007/978-3-662-47666-6\_38.
- 5 Dan Alistarh, Martin Töpfer, and Przemysław Uznański. Robust comparison in population protocols, 2020. arXiv:2003.06485.
- 6 Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, pages 235–253, March 2006.
- 7 Dana Angluin, James Aspnes, and David Eisenstat. A simple population protocol for fast robust approximate majority. *Distributed Computing*, 21(2):87–102, 2008.
- 8 Luca Becchetti, Andrea Clementi, and Emanuele Natale. Consensus dynamics: An overview. *ACM SIGACT News*, 51(1):58–104, 2020.
- 9 Luca Cardelli and Attila Csikász-Nagy. The cell cycle switch computes approximate majority. *Scientific reports*, 2:656, September 2012. doi:10.1038/srep00656.
- 10 Ho-Lin Chen, David Doty, and David Soloveichik. Deterministic function computation with chemical reaction networks. *Nat. Comput.*, 13(4):517–534, 2014. doi:10.1007/s11047-013-9393-6.
- 11 Yuan-Jyue Chen, Neil Dalchau, Niranjan Srinivas, Andrew Phillips, Luca Cardelli, David Soloveichik, and Georg Seelig. Programmable chemical controllers made from dna. *Nature nanotechnology*, 8, September 2013. doi:10.1038/nnano.2013.189.
- 12 Anne Condon, Monir Hajiaghayi, David Kirkpatrick, and Ján Maňuch. Approximate majority analyses using tri-molecular chemical reaction networks. *Natural Computing*, pages 1–22, 2019.
- 13 Francesco d’Amore, Andrea E. F. Clementi, and Emanuele Natale. Phase transition of a non-linear opinion dynamics with noisy interactions - (extended abstract). In Andrea Werneck Richa and Christian Scheideler, editors, *Structural Information and Communication Complexity - 27th International Colloquium, SIROCCO 2020, Paderborn, Germany, June 29 - July 1, 2020, Proceedings*, volume 12156 of *Lecture Notes in Computer Science*, pages 255–272. Springer, 2020. doi:10.1007/978-3-030-54921-3\_15.
- 14 Bartłomiej Dudek and Adrian Kosowski. Universal protocols for information dissemination using emergent signals. In Ilias Diakonikolas, David Kempe, and Monika Henzinger, editors, *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, June 25-29, 2018*, pages 87–99. ACM, 2018. doi:10.1145/3188745.3188818.



- 15 William Feller. *An introduction to probability theory and its applications. Vol. I.* Third edition. John Wiley & Sons Inc., New York, 1968.
- 16 Leszek Gasieniec and Grzegorz Stachowiak. Fast space optimal leader election in population protocols. In Artur Czumaj, editor, *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 2653–2667. SIAM, 2018. doi:10.1137/1.9781611975031.169.
- 17 Leszek Gasieniec, Grzegorz Stachowiak, and Przemyslaw Uznanski. Almost logarithmic-time space optimal leader election in population protocols. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '19*, pages 93–102, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3323165.3323178.
- 18 Geoffrey R Grimmett and David R Stirzaker. *Probability and Random Processes.* Oxford University Press, 2001.
- 19 Adrian Kosowski and Przemysław Uznański. Population protocols are fast. *arXiv preprint arXiv:1802.06872*, 2018.
- 20 Etienne Perron, Dinkar Vasudevan, and Milan Vojnovic. Using three states for binary consensus on complete graphs. In *IEEE INFOCOM 2009*, pages 2527–2535. IEEE, 2009.
- 21 Chris Thachuk, Erik Winfree, and David Soloveichik. Leakless DNA strand displacement systems. In Andrew Phillips and Peng Yin, editors, *DNA Computing and Molecular Programming - 21st International Conference, DNA 21, Boston and Cambridge, MA, USA, August 17-21, 2015. Proceedings*, volume 9211 of *Lecture Notes in Computer Science*, pages 133–153. Springer, 2015. doi:10.1007/978-3-319-21999-8\_9.
- 22 Boya Wang, Chris Thachuk, Andrew D. Ellington, Erik Winfree, and David Soloveichik. Effective design principles for leakless strand displacement systems. *Proceedings of the National Academy of Sciences*, 115(52):E12182–E12191, 2018. doi:10.1073/pnas.1806859115.
- 23 A. C. Yao. Probabilistic computations: Toward a unified measure of complexity. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 222–227, 1977.
- 24 Ercan Yildiz, Asuman Ozdaglar, Daron Acemoglu, Amin Saberi, and Anna Scaglione. Binary opinion dynamics with stubborn agents. *ACM Transactions on Economics and Computation (TEAC)*, 1(4):1–30, 2013.


# Distributed Runtime Verification Under Partial Synchrony

Ritam Ganguly

Michigan State University, East Lansing, MI, USA  
gangulyr@msu.edu

Anik Momtaz 

Michigan State University, East Lansing, MI, USA  
momtazan@msu.edu

Borzoo Bonakdarpour 

Michigan State University, East Lansing, MI, USA  
borzoo@msu.edu

---

## Abstract

In this paper, we study the problem of runtime verification of distributed applications that do *not* share a global clock with respect to specifications in the linear temporal logics (LTL). Our proposed method distinguishes from the existing work in three novel ways. First, we make a practical assumption that the distributed system under scrutiny is augmented with a clock synchronization algorithm that guarantees bounded clock skew among all processes. Second, we do not make any assumption about the structure of predicates that form LTL formulas. This relaxation allows us to monitor a wide range of applications that was not possible before. Subsequently, we propose a distributed monitoring algorithm by employing SMT solving techniques. Third, given the fact that distributed applications nowadays run on massive cloud services, we extend our solution to a parallel monitoring algorithm to utilize the available computing infrastructure. We report on rigorous synthetic as well as real-world case studies and demonstrate that scalable online monitoring of distributed applications is within our reach.

**2012 ACM Subject Classification** Theory of computation → Logic and verification; Theory of computation → Distributed computing models

**Keywords and phrases** Runtime monitoring, Distributed systems, Formal methods, Cassandra

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2020.20

**Funding** *Borzoo Bonakdarpour*: This work is partially sponsored by the NSF FMitF Award 1917979.

## 1 Introduction

A *distributed system* consists of a collection of processes that attempt to solve a problem by means of communication and local computation. Applications of distributed systems range over small-scale networks of deeply embedded systems to monitoring a collection of sensors in smart buildings to large-scale cluster of servers in cloud services. However, design and analysis of such systems has always been a grand challenge due to their inherent complex structure, amplified by combinatorial explosion of possible executions due to nondeterminism and the occurrence of faults. This makes exhaustive model checking techniques not scalable and under-approximate techniques such as testing not so effective.

In this paper, we advocate for a *runtime verification* (RV) approach, where a monitor observes the behavior of a distributed system at run time and verifies its correctness with respect to a temporal logic formula. Distributed RV has to overcome a significant challenge. Although RV deals with finite executions, due to lack of a global clock, there may potentially exist events whose order of occurrence cannot be determined by a runtime monitor. Additionally, different orders of events may result in different verification verdicts. Enumerating all



© Ritam Ganguly, Anik Momtaz, and Borzoo Bonakdarpour;  
licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

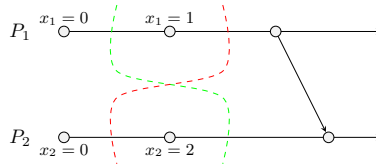
Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 20; pp. 20:1–20:17

Leibniz International Proceedings in Informatics



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

possible orders at run time often incurs an exponential blow up, making it impractical. This is of course, on top of the usual monitor overhead to evaluate an execution. For example, consider the distributed computation in Fig. 1, where processes  $P_1$  and  $P_2$  host discrete variables  $x_1$  and  $x_2$ , respectively. Let us also consider LTL formula  $\varphi = \bigcirc(x_1 + x_2 \leq 1)$ . Since events  $x_1 = 1$  and  $x_2 = 2$  are *concurrent* (i.e., it is not possible to determine which happened before or after which in the absence of a global clock), the formula can be evaluated to both true and false, depending upon different order of occurrences of these events. Handling concurrent events generally results in combinatorial enumeration of all possibilities and, hence, intractability of distributed RV. Existing distributed RV techniques operate in two extremes: they either assume a global clock [1], which is unrealistic for large-scale distributed settings or assume complete asynchrony [20, 19], which do not scale well.



■ **Figure 1** Distributed computation.

We propose a sound and complete solution to the problem of distributed RV with respect to LTL formulas by incorporating a middle-ground approach. Our solution uses a fault-proof central monitor and may be summarized as follows. In order to remedy the explosion of different interleavings, we make a practical assumption, that is, a *bounded skew*  $\epsilon$  between local clocks of every pair of processes, guaranteed by a fault-proof clock synchronization algorithm (e.g., NTP [17]). This means time instants from different clocks within  $\epsilon$  are considered concurrent, i.e., it is not possible to determine their order of occurrence. This setting constitutes *partial synchrony*, which does not assume a global clock but limits the impact of asynchrony within clock drifts. Following the work in [14], we augment the classic *happened-before* relation [16] with the bounded skew assumption. This way, concurrent events are limited to those that happen within the  $\epsilon$  time window, and those cannot be ordered according to communication. We transform our monitoring decision problem into an SMT solving problem. The SMT instance includes constraints that encode (1) our monitoring algorithm based on the 3-valued semantics of LTL [2], (2) behavior of communicating processes and their local state changes in terms of a distributed computation, and (3) the happened-before relation subject to the  $\epsilon$  clock skew assumption. Then, it attempts to concretize an uninterpreted function whose evaluation provides the possible verdicts of the monitor with respect to the given computation. Furthermore, given the fact that distributed applications nowadays run on massive cloud services, we extend our solution to a parallel monitoring algorithm to utilize the available computing infrastructure and achieve better scalability.

We have fully implemented our techniques and report results of rigorous experiments on monitoring synthetic data, as well as monitoring consistency conditions in data centers that run Cassandra [15] as their distributed database management system. We make the following observations. First, although our approach is based on SMT solving, it can be employed for offline monitoring (e.g., log analysis) as well as online monitoring for less intensive applications such as consistency checking in Google Drive. Secondly, we show how the structure of global predicates (e.g., conjunctive vs. disjunctive) and LTL formulas affect the performance of monitoring. Third, we illustrate how monitoring overhead is *independent* of the clock skews when practical clock synchronization protocols are applied, making the drift sufficiently small. Finally, we demonstrate how our parallel monitoring algorithm achieves scalability, especially for predicate detection.

*Organization.* Section 2 presents the background concepts. Our SMT-based solution is described in Section 3, while experimental results are analyzed in Section 4. Related work is discussed in Section 5. Finally, we make concluding remarks in Section 6.

## 2 Preliminaries

### 2.1 Linear Temporal Logic (LTL) for RV

Let  $AP$  be a set of *atomic propositions* and  $\Sigma = 2^{AP}$  be the set of all possible *states*. A *trace* is a sequence  $s_0s_1\cdots$ , where  $s_i \in \Sigma$  for every  $i \geq 0$ . We denote by  $\Sigma^*$  (resp.,  $\Sigma^\omega$ ) the set of all finite (resp., infinite) traces. The syntax and semantics of the *linear temporal logic* (LTL) [21] are defined for infinite traces. The syntax is defined by the following grammar:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc\varphi \mid \varphi \mathcal{U} \varphi$$

where  $p \in AP$ , and where  $\bigcirc$  and  $\mathcal{U}$  are the “next” and “until” temporal operators respectively. We also use the following abbreviations: **true** =  $p \vee \neg p$ , **false** =  $\neg$ **true**,  $\varphi \rightarrow \psi = \neg\varphi \vee \psi$ ,  $\varphi \wedge \psi = \neg(\neg\varphi \vee \neg\psi)$ ,  $\diamond\varphi = \mathbf{true} \mathcal{U} \varphi$  (*eventually*  $\varphi$ ), and  $\square\varphi = \neg\diamond\neg\varphi$  (*always*  $\varphi$ ).

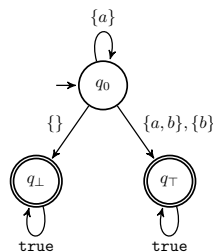
The infinite-trace semantics of LTL is defined as follows. Let  $\sigma = s_0s_1s_2\cdots \in \Sigma^\omega$ ,  $i \geq 0$ , and let  $\models$  denote the *satisfaction* relation:

$$\begin{array}{lll} \sigma, i \models p & \text{iff} & p \in s_i \\ \sigma, i \models \neg\varphi & \text{iff} & \sigma, i \not\models \varphi \\ \sigma, i \models \varphi_1 \vee \varphi_2 & \text{iff} & \sigma, i \models \varphi_1 \text{ or } \sigma, i \models \varphi_2 \\ \sigma, i \models \bigcirc\varphi & \text{iff} & \sigma, i+1 \models \varphi \\ \sigma, i \models \varphi_1 \mathcal{U} \varphi_2 & \text{iff} & \exists k \geq i. \sigma, k \models \varphi_2 \text{ and } \forall j \in [i, k) : \sigma, j \models \varphi_1 \end{array}$$

Also,  $\sigma \models \varphi$  holds if and only if  $\sigma, 0 \models \varphi$  holds.

In the context of RV, the 3-valued LTL ( $LTL_3$  for short) [2] evaluates LTL formulas for *finite* traces, but with an eye on possible future extensions. In  $LTL_3$ , the set of truth values is  $\mathbb{B}_3 = \{\top, \perp, ?\}$ , where  $\top$  (resp.,  $\perp$ ) denotes that the formula is *permanently* satisfied (resp., violated), no matter how the current finite trace extends, and “?” denotes an *unknown* verdict, i.e., there exists an extension that can violate the formula, and another extension that can satisfy the formula. Let  $\alpha \in \Sigma^*$  be a non-empty finite trace. The truth value of an  $LTL_3$  formula  $\varphi$  with respect to  $\alpha$ , denoted by  $[\alpha \models_3 \varphi]$ , is defined as follows:

$$[\alpha \models_3 \varphi] = \begin{cases} \top & \text{if } \forall \sigma \in \Sigma^\omega : \alpha\sigma \models \varphi \\ \perp & \text{if } \forall \sigma \in \Sigma^\omega : \alpha\sigma \not\models \varphi \\ ? & \text{otherwise.} \end{cases}$$



■ **Figure 2**  $LTL_3$  monitor for  $\varphi = a \mathcal{U} b$ .

For example, consider formula  $\varphi = \Box p$ , and a finite trace  $\alpha = s_0 s_1 \cdots s_n$ . If  $p \notin s_i$  for some  $i \in [0, n]$ , then  $[\alpha \models_3 \varphi] = \perp$ , that is, the formula is permanently violated. Now, consider formula  $\varphi = \Diamond p$ . If  $p \notin s_i$  for all  $i \in [0, n]$ , then  $[\alpha \models_3 \varphi] = ?$ .

► **Definition 1.** *The LTL<sub>3</sub> monitor (see Fig. 2) for a formula  $\varphi$  is the unique deterministic finite state machine  $\mathcal{M}_\varphi = (\Sigma, Q, q_0, \delta, \lambda)$ , where  $Q$  is the set of states,  $q_0$  is the initial state,  $\delta : Q \times \Sigma \rightarrow Q$  is the transition function, and  $\lambda : Q \rightarrow \mathbb{B}_3$  is a function such that  $\lambda(\delta(q_0, \alpha)) = [\alpha \models_3 \varphi]$ , for every finite trace  $\alpha \in \Sigma^*$ .*

## 2.2 Distributed Computations

We assume a loosely coupled asynchronous message passing system, consisting of  $n$  reliable processes (that do not fail), denoted by  $\mathcal{P} = \{P_1, P_2, \dots, P_n\}$ , without any shared memory or global clock. Channels are assumed to be FIFO, and lossless. In our model, each local state change is considered an event, and every message activity (send or receive) is also represented by a new event. Message transmission does not change the local state of processes and the content of a message is immaterial to our purposes. We will need to refer to some global clock which acts as a “real” timekeeper. It is to be understood, however, that this global clock is a theoretical object used in definitions, and is *not* available to the processes.

We make a practical assumption, known as *partial synchrony*. The *local clock* (or time) of a process  $P_i$ , where  $i \in [1, n]$ , can be represented as an increasing function  $c_i : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}_{\geq 0}$ , where  $c_i(\chi)$  is the value of the local clock at global time  $\chi$ . Then, for any two processes  $P_i$  and  $P_j$ , we have  $\forall \chi \in \mathbb{R}_{\geq 0}. |c_i(\chi) - c_j(\chi)| < \epsilon$ , with  $\epsilon > 0$  being the maximum *clock skew*. The value  $\epsilon$  is assumed to be fixed and known by the monitor in the rest of this paper. In the sequel, we make it explicit when we refer to “local” or ‘global’ time. This assumption is met by using a clock synchronization algorithm, like NTP [17], to ensure bounded clock skew among all processes.

An *event* in process  $P_i$  is of the form  $e_{\tau, \sigma}^i$ , where  $\sigma$  is *logical time* (i.e., a natural number) and  $\tau$  is the local time at global time  $\chi$ , that is,  $\tau = c_i(\chi)$ . We assume that for every two events  $e_{\tau, \sigma}^i$  and  $e_{\tau', \sigma'}^i$ , we have  $(\tau < \tau') \Leftrightarrow (\sigma < \sigma')$ .

► **Definition 2.** *A distributed computation on  $N$  processes is a tuple  $(\mathcal{E}, \rightsquigarrow)$ , where  $\mathcal{E}$  is a set of events partially ordered by Lamport’s happened-before ( $\rightsquigarrow$ ) relation [16], subject to the partial synchrony assumption:*

■ *In every process  $P_i$ ,  $1 \leq i \leq N$ , all events are totally ordered, that is,*

$$\forall \tau, \tau' \in \mathbb{R}_+. \forall \sigma, \sigma' \in \mathbb{Z}_{\geq 0}. (\sigma < \sigma') \rightarrow (e_{\tau, \sigma}^i \rightsquigarrow e_{\tau', \sigma'}^i).$$

- *If  $e$  is a message send event in a process, and  $f$  is the corresponding receive event by another process, then we have  $e \rightsquigarrow f$ .*
- *For any two processes  $P_i$  and  $P_j$ , and any two events  $e_{\tau, \sigma}^i, e_{\tau', \sigma'}^j \in \mathcal{E}$ , if  $\tau + \epsilon < \tau'$ , then  $e_{\tau, \sigma}^i \rightsquigarrow e_{\tau', \sigma'}^j$ , where  $\epsilon$  is the maximum clock skew.*
- *If  $e \rightsquigarrow f$  and  $f \rightsquigarrow g$ , then  $e \rightsquigarrow g$ .*

► **Definition 3.** *Given a distributed computation  $(\mathcal{E}, \rightsquigarrow)$ , a subset of events  $C \subseteq \mathcal{E}$  is said to form a consistent cut iff when  $C$  contains an event  $e$ , then it contains all events that happened-before  $e$ . Formally,  $\forall e \in \mathcal{E}. (e \in C) \wedge (f \rightsquigarrow e) \rightarrow f \in C$ .*

The *frontier* of a consistent cut  $C$ , denoted  $\text{front}(C)$  is the set of events that happen last in the cut.  $\text{front}(C)$  is a set of  $e_{last}^i$  for each  $i \in [1, |\mathcal{P}|]$  and  $e_{last}^i \in C$ . We denote  $e_{last}^i$  as the last event in  $P_i$  such that  $\forall e_{\tau, \sigma}^i \in \mathcal{E}. (e_{\tau, \sigma}^i \neq e_{last}^i) \rightarrow (e_{\tau, \sigma}^i \rightsquigarrow e_{last}^i)$ .

## 2.3 Problem Statement

Given a distributed computation  $(\mathcal{E}, \rightsquigarrow)$ , a *valid* sequence of consistent cuts is of the form  $C_0C_1C_2\cdots$ , where for all  $i \geq 0$ , we have (1)  $C_i \subset C_{i+1}$ , and (2)  $|C_i| + 1 = |C_{i+1}|$ . Let  $\mathcal{C}$  denote the set of all valid sequences of consistent cuts. We define the set of all traces of  $(\mathcal{E}, \rightsquigarrow)$  as follows:

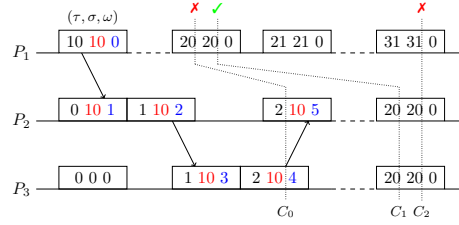
$$\text{Tr}(\mathcal{E}, \rightsquigarrow) = \left\{ \text{front}(C_0)\text{front}(C_1)\cdots \mid C_0C_1C_2\cdots \in \mathcal{C} \right\}.$$

Now, the evaluation of an LTL formula  $\varphi$  with respect to  $(\mathcal{E}, \rightsquigarrow)$  in the 3-valued semantics is the following:

$$[(\mathcal{E}, \rightsquigarrow) \models_3 \varphi] = \left\{ (\alpha, \rightsquigarrow) \models_3 \varphi \mid \alpha \in \text{Tr}(\mathcal{E}, \rightsquigarrow) \right\}$$

This means evaluating a distributed computation with respect to a formula results in a *set* of verdicts, as a computation may involve several traces.

## 2.4 Hybrid Logical Clocks



■ **Figure 3** HLC example.

A *hybrid logical clock* (HLC) [14] is a tuple  $(\tau, \sigma, \omega)$  for detecting one-way causality, where  $\tau$  is the local time,  $\sigma$  ensures the order of send and receive events between two processes, and  $\omega$  indicates causality between events. Thus, in the sequel, we denote an event by  $e_{\tau, \sigma, \omega}^i$ . More specifically, for a set  $\mathcal{E}$  of events:

- $\tau$  is the local clock value of events, where for any process  $P_i$  and two events  $e_{\tau, \sigma, \omega}^i, e_{\tau', \sigma', \omega'}^i \in \mathcal{E}$ , we have  $\tau < \tau'$  iff  $e_{\tau, \sigma, \omega}^i \rightsquigarrow e_{\tau', \sigma', \omega'}^i$ .
- $\sigma$  stipulates the logical time, where:
  - For any process  $P_i$  and any event  $e_{\tau, \sigma, \omega}^i \in \mathcal{E}$ ,  $\tau$  never exceeds  $\sigma$ , and their difference is bounded by  $\epsilon$  (i.e.,  $\sigma - \tau \leq \epsilon$ ).
  - For any two processes  $P_i$  and  $P_j$ , and any two events  $e_{\tau, \sigma, \omega}^i, e_{\tau', \sigma', \omega'}^j \in \mathcal{E}$ , where event  $e_{\tau, \sigma, \omega}^i$  receiving a message sent by event  $e_{\tau', \sigma', \omega'}^j$ ,  $\sigma$  is updated to  $\max\{\sigma, \sigma', \tau\}$ . The maximum of the three values are chosen to ensure that  $\sigma$  remains updated with the largest  $\tau$  observed so far. Observe that  $\sigma$  has similar behavior as  $\tau$ , except the communication between processes has no impact on the value of  $\tau$  for an event.
- $\omega : \mathcal{E} \rightarrow \mathbb{Z}_{\geq 0}$  is a function that maps each event in  $\mathcal{E}$  to the causality updates, where:
  - For any process  $P_i$  and a send or local event  $e_{\tau, \sigma, \omega}^i \in \mathcal{E}$ , if  $\tau < \sigma$ , then  $\omega$  is incremented. Otherwise,  $\omega$  is reset to 0.
  - For any two processes  $P_i$  and  $P_j$  and any two events  $e_{\tau, \sigma, \omega}^i, e_{\tau', \sigma', \omega'}^j \in \mathcal{E}$ , where event  $e_{\tau, \sigma, \omega}^i$  receiving a message sent by event  $e_{\tau', \sigma', \omega'}^j$ ,  $\omega(e_{\tau, \sigma, \omega}^i)$  is updated based on  $\max\{\sigma, \sigma', \tau\}$ .
  - For any two processes  $P_i$  and  $P_j$ , and any two events  $e_{\tau, \sigma, \omega}^i, e_{\tau', \sigma', \omega'}^j \in \mathcal{E}$ ,  $(\tau = \tau') \wedge (\omega < \omega') \rightarrow e_{\tau, \sigma, \omega}^i \rightsquigarrow e_{\tau', \sigma', \omega'}^j$ .

In our implementation of HLC, we assume that it is fault-proof. Fig. 3 shows an HLC incorporated partially synchronous concurrent timelines of three processes with  $\varepsilon = 10$ . Observe that the local times of all events in  $\text{front}(C_1)$  are bounded by  $\varepsilon$ . Therefore,  $C_1$  is a consistent cut, but  $C_0$  and  $C_2$  are not.

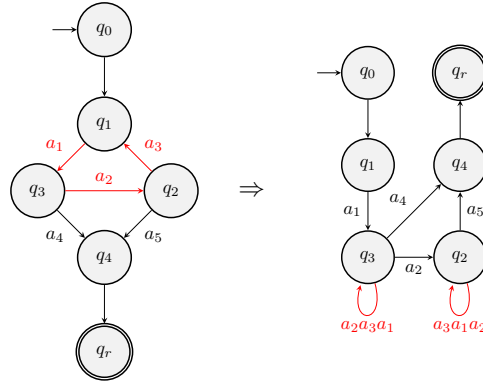
### 3 SMT-based Solution

#### 3.1 Overall Idea

Recall from Section 1 (Fig. 1) that monitoring a distributed computation may result in multiple verdicts depending upon different ordering of events. In other words, given a distributed computation  $(\mathcal{E}, \rightsquigarrow)$  and an LTL formula  $\varphi$ , different ordering of events may reach different states in the monitor automaton  $\mathcal{M}_\varphi = (\Sigma, Q, q_0, \delta, \lambda)$  (as defined in Definition 1). In order to ensure that all possible verdicts are explored, we generate an SMT instance for (1) the distributed computation  $(\mathcal{E}, \rightsquigarrow)$ , and (2) each possible path in the LTL<sub>3</sub> monitor. Thus, the corresponding decision problem is the following: given  $(\mathcal{E}, \rightsquigarrow)$  and a monitor path  $q_0 q_1 \cdots q_m$  in an LTL<sub>3</sub> monitor, can  $(\mathcal{E}, \rightsquigarrow)$  reach  $q_m$ ? If the SMT instance is satisfiable, then  $\lambda(q_m)$  is a possible verdict. For example, for the monitor in Fig. 2, we consider two paths  $q_0^* q_{\perp}$  and  $q_0^* q_{\top}$  (and, hence, two SMT instances). Thus, if both instances turn out to be unsatisfiable, then the resulting monitor state is  $q_0$ , where  $\lambda(q_0) = ?$ .

We note that since LTL<sub>3</sub> monitors may contain cycles, we first transform the monitor into an acyclic monitor. To this end, we collapse each cycle into one state with a self-loop labeled by the sequence of events on the cycle (see Fig. 4 for an example). In the next two subsections, we present the SMT entities and constraints with respect to *one* monitor path and a distributed computation.

#### 3.2 SMT Entities



■ Figure 4 LTL<sub>3</sub> Monitor cycle.

We now introduce the entities that represent a path in an LTL<sub>3</sub> monitor  $\mathcal{M}_\varphi = (\Sigma, Q, q_0, \delta, \lambda)$  for LTL formula  $\varphi$  and computation  $(\mathcal{E}, \rightsquigarrow)$ .

**Monitor automaton.** Let  $q_0 \xrightarrow{s_0} q_1 \xrightarrow{s_1} \cdots (q_j \xrightarrow{s_j} q_j)^* \cdots \xrightarrow{s_{m-1}} q_m$  be a path of monitor  $\mathcal{M}_\varphi$ , which may or may not include a self-loop. We include a non-negative integer variable  $k_i$  for each transition  $q_i \xrightarrow{s_i} q_{i+1}$ , where  $i \in [0, m-1]$  and  $s_i \in \Sigma$ . Observe that we include only one non-negative integer variable  $k_j$  for the self-loop  $q_j \xrightarrow{s_j} q_j$ .



**Distributed computation.** In our SMT encoding, we represent the set  $\mathcal{E}$  by a bit-vector for efficiency. However, for simplicity, we keep referring to the events in a distributed computation by the set  $\mathcal{E}$ . In order to express the happened-before relation in our SMT encoding, we conduct a pre-processing phase, where we create an  $|\mathcal{E}| \times |\mathcal{E}|$  matrix  $E$ , such that  $E[i, j] = 1$ , if  $E[i] \rightsquigarrow E[j]$ , else  $E[i, j] = 0$ . This pre-processing phase incorporates the HLC algorithm, described in Section 2.4, to construct the matrix. In the sequel, for simplicity, we keep using the  $\rightsquigarrow$  relation between events when needed.

In order to establish the connection between events and atomic propositions in AP based on which the LTL formula  $\varphi$  is constructed, we introduce a Boolean function  $\mu : \mathcal{E} \times \Sigma \rightarrow \{\mathbf{true}, \mathbf{false}\}$ . We note that if processes have non-Boolean variables and more complex relational predicates (e.g.,  $x_1 + x_2 \geq 2$ ), then function  $\mu$  can be defined accordingly. Finally, in order to identify the sequence of consistent cuts whose run on the monitor starts from  $q_0$  and ends in  $q_m$ , we introduce an *uninterpreted* function  $\rho : \mathbb{Z}_{\geq 0} \rightarrow 2^{\mathcal{E}}$ . That is, if the SMT instance is satisfiable, then the interpretation of  $\rho$  is the sequence of consistent cuts that ends in monitor state  $q_m$ . Otherwise, no ordering of concurrent events results in the verdict given by state  $q_m$ .

### 3.3 SMT Constraints

Once we define the necessary SMT entities, we move onto the SMT constraints.

**Consistent cut constraints over  $\rho$ .** We first identify the constraints over uninterpreted function  $\rho$ , whose interpretation is a sequence of consistent cuts that starts and ends in the given monitor automaton path. Thus, we first require that each element in the range of  $\rho$  must be a consistent cut:

$$\forall i \in [0, m]. \forall e, e' \in \mathcal{E}. \left( (e' \rightsquigarrow e) \wedge (e \in \rho(i)) \right) \rightarrow (e' \in \rho(i))$$

Next, we require the sequence of consistent cuts that  $\rho$  identifies to start from an empty set of events and in each consistent cut of the sequence, there is one more event in the successor cut:

$$\forall i \in [0, m]. |\rho(i+1)| = |\rho(i)| + 1$$

Finally, the progression of consistent cuts should yield a subset relation. Otherwise, the successor of a consistent cut is not an immediately reachable cut in  $(\mathcal{E}, \rightsquigarrow)$ :

$$\forall i \in [0, m]. \rho(i) \subseteq \rho(i+1)$$

**Monitoring constraints over  $\rho$ .** These constraints are responsible for generating a valid sequence of consistent cuts given a distributed computation  $(\mathcal{E}, \rightsquigarrow)$  that runs on monitor path  $q_1 \xrightarrow{s_1} q_2 \cdots q_j^* \cdots \xrightarrow{s_{m-1}} q_m$ . We begin with interpreting  $\rho(k_m)$  by requiring that running  $(\mathcal{E}, \rightsquigarrow)$  ends in monitor state  $q_m$ . The corresponding SMT constraint is:

$$\mu(\text{front}(\rho(k_m)), s_{m-1})$$

For every monitor state  $q_i$ , where  $i \in [0, m-1]$ , if  $q_i$  does not have a self-loop, the corresponding SMT constraint is:

$$\mu(\text{front}(\rho(k_{i+1} - 1)), s_i) \wedge (k_i = k_{i+1} - 1)$$

For every monitor state  $q_j$ , where  $j \in [0, m - 1]$ , suppose  $q_j$  has a self-loop (recall that a cycle of  $r$  transitions in the monitor automaton is collapsed into a self-loop labeled by a sequence of  $r$  letters). Let us imagine that this self-loop executed  $z$  number of times for some  $z \geq 0$ . Furthermore, we denote the sequence of letters in the self-loop as  $s_{j_1} s_{j_2} \cdots s_{j_r}$ . The corresponding SMT constraint is:

$$\bigwedge_{i=1}^z \bigwedge_{n=1}^r \mu(\text{front}(\rho(k_j + r(i-1) + n)), s_{j_n})$$

Again, since  $z$  is a free variable in the above constraint, the solver will identify some value  $z \geq 0$  which is exactly what we need. To ensure that the domain of  $\rho$  starts from the empty consistent cut (i.e.,  $\rho(0) = \emptyset$ ), we add:

$$k_0 = 0.$$

Finally, let  $C$  denote the conjunction of all the above constraints. Recall that this conjunction is with respect to only one monitor path from  $q_0$  to  $q_m$ . Since there may be multiple paths in the monitor automaton that can reach  $q_m$  from  $q_0$ , we replicate the above constraints for each such path. Suppose there are  $n$  such paths and let  $C_1, C_2, \dots, C_n$  be the corresponding SMT constraints for these  $n$  paths. We include the following constraint:

$$C_1 \vee C_2 \vee C_3 \vee \cdots \vee C_n$$

This means that if the SMT instance is satisfiable, then computation  $(\mathcal{E}, \rightsquigarrow)$  can reach monitor state  $q_m$  from  $q_0$ .

### 3.4 Segmentation of Distributed Computation

Since the RV problem is known to be NP-complete in the size of processes [9], we are inherently dealing with a computationally difficult problem. This complexity also grows to higher classes in the presence of nested temporal operators. In order to cope with this complexity, our strategy is to chop a computation  $(\mathcal{E}, \rightsquigarrow)$  into a sequence of small *segments*  $(seg_1, \rightsquigarrow)(seg_2, \rightsquigarrow) \cdots (seg_g, \rightsquigarrow)$  to create more but smaller-size SMT problems. This is likely to improve the overall performance dramatically. More specifically, in a computation whose duration is  $l$ , for  $g$  number of segments (i.e., segment duration  $\frac{l}{g} \pm \epsilon$ ), the set of events in segment  $j$ , where  $j \in [1, g]$ , is the following:

$$seg_j = \left\{ e_{\tau, \sigma, \omega}^n \mid \sigma \in [\max\{0, \frac{(j-1)l}{g} - \epsilon\}, \frac{jl}{g}] \wedge n \in [1, |\mathcal{P}|] \right\}$$

Observe that monitoring a segment has to be conducted from  $\epsilon$  time units before the segment actually starts. Also, when monitoring segment  $j$  is concluded, monitoring segment  $j + 1$  should start from all possible monitor states that can be reached by segment  $j$ . In Section 4, we show the impact of segmentation on the overall performance of monitoring.

We now show that the verification of a sequence of segments of a distributed computation results in the same set of verdict as verification of the computation in one shot. This can be formally proved by construction as follows. Given  $(\mathcal{E}, \rightsquigarrow)$  and  $\varphi$ , where  $(\mathcal{E}, \rightsquigarrow)$  is chopped into two segments  $(seg_1, \rightsquigarrow)$  and  $(seg_2, \rightsquigarrow)$ , we have:  $[(\mathcal{E}, \rightsquigarrow) \models_3 \varphi] = [(seg_1 seg_2, \rightsquigarrow) \models_3 \varphi]$ . Let  $Q_1$  be the set of all reachable monitor states at the end of verifying  $(seg_1, \rightsquigarrow)$ . This set represents the valuation of  $(seg_1, \rightsquigarrow)$  with respect to  $\varphi$ . Since in our algorithm verification of  $(seg_2, \rightsquigarrow)$  starts with states in  $Q_1$  as initial states of the monitor, we do not lose the temporal

order of events. In other words,  $Q_1$  encodes all the important observations in  $(seg_1, \rightsquigarrow)$ . This implies that by construction, the set  $Q_2$  of reachable monitor states after verification of  $(seg_2, \rightsquigarrow)$  starting from  $Q_1$  is the set of all reachable monitor states when verifying  $(\mathcal{E}, \rightsquigarrow)$ . By induction, the same can be proved for  $g$  segments.

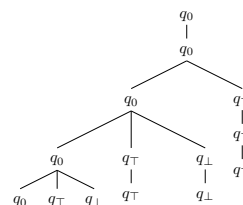
### 3.5 Parallelized Monitoring

We parallelize our technique in two steps, which ensure the temporal order of events. Let  $\mathcal{M}_\varphi = (\Sigma, Q, q_0, \delta, \lambda)$  be an  $\text{LTL}_3$  monitor. Our first step is to create a 3-dimensional reachability matrix  $RM$  by solving the following SMT decision problem: given a current monitor state  $q_j \in Q$  and segment  $seg_i$ , can this segment reach monitor state  $q_k \in Q$ , for all  $i \in [1, g]$ , and  $j, k \in [0, |Q| - 1]$ . If the answer to the problem is affirmative, then we mark  $RM[i][j][k]$  with **true**, otherwise with **false**. This is illustrated in Fig. 5 for the monitor shown in Fig. 2, where the grey cells are filled arbitrarily with the answer to the SMT problem. This step can be made embarrassingly parallel, where each element of  $RM$  can be computed independently by a different computing core. One can optimize the construction of  $RM$  by omitting redundant SMT executions. For example, if  $RM[i][j][\top] = \text{true}$ , then  $RM[i'][\top][\top] = \text{true}$  for all  $i' \in [i, |Q| - 1]$ . Likewise, if  $RM[i][j][\perp] = \text{true}$ , then  $RM[i'][\perp][\perp] = \text{true}$  for all  $i' \in [i, |Q| - 1]$ .

The second step is to generate a verdict reachability tree from  $RM$ . The goal of the tree is to check if a monitor state  $q_m \in Q$  can be reached from the initial monitor state  $q_0$ . This is achieved by setting  $q_0$  as the root and generating all possible paths from  $q_0$  using  $RM$ . That is, if  $RM[i][k][j] = \text{true}$ , then we create a tree node with label  $q_j$  and add it as a child of the node with the label  $q_k$ . Once the tree is generated, if  $q_m$  is one of the leaves, only then we can say  $q_m$  is reachable from  $q_0$ . In general, all leaves of the tree are possible monitoring verdicts. Note that creation of the tree is achieved using a sequential algorithm. For example, Fig.6 shows the verdict reachability tree generated from the matrix in Fig. 5.

	seg <sub>1</sub>			seg <sub>2</sub>			seg <sub>3</sub>			seg <sub>4</sub>		
q <sub>0</sub>	q <sub>0</sub>	q <sub>⊤</sub>	q <sub>⊥</sub>	q <sub>0</sub>	q <sub>⊤</sub>	q <sub>⊥</sub>	q <sub>0</sub>	q <sub>⊤</sub>	q <sub>⊥</sub>	q <sub>0</sub>	q <sub>⊤</sub>	q <sub>⊥</sub>
	T	F	F	T	T	F	T	T	T	T	T	T
q <sub>⊤</sub>	q <sub>0</sub>	q <sub>⊤</sub>	q <sub>⊥</sub>	q <sub>0</sub>	q <sub>⊤</sub>	q <sub>⊥</sub>	q <sub>0</sub>	q <sub>⊤</sub>	q <sub>⊥</sub>	q <sub>0</sub>	q <sub>⊤</sub>	q <sub>⊥</sub>
	F	F	F	F	T	F	F	T	F	F	T	F
q <sub>⊥</sub>	q <sub>0</sub>	q <sub>⊤</sub>	q <sub>⊥</sub>	q <sub>0</sub>	q <sub>⊤</sub>	q <sub>⊥</sub>	q <sub>0</sub>	q <sub>⊤</sub>	q <sub>⊥</sub>	q <sub>0</sub>	q <sub>⊤</sub>	q <sub>⊥</sub>
	F	F	F	F	F	T	F	F	F	T	F	F

■ **Figure 5** Reachability Matrix for  $aUb$ .



■ **Figure 6** Reachability Tree for  $aUb$ .

## 4 Case Studies and Evaluation

In this section, we evaluate our technique using synthetic experiments and a case study involving Cassandra, a distributed database <sup>1</sup>. We emphasize although RV involves many dimensions such as instrumentation, data collection, data transfer to the monitor, etc., our goal in this section is to evaluate our SMT-based technique, as in a distributed setting, the analysis time is the dominant factor over other types of overhead.

<sup>1</sup> All experimental code and data is available at <https://drive.google.com/file/d/191F-jfUXV-18ssxuRli1sixw2vctmofA/view?usp=sharing>

## 4.1 Implementation and Experimental Setup

Each experiment in this section consists of two phases: (1) data collection, and (2) verification. We developed a program that randomly generates a distributed computation (i.e., the behavior of a set of processes in terms of their local events and communication). We use a uniform distribution  $(0, 2)$  to define the type of the event (computation, send, receive). Then another program observes the execution of these processes and generates a trace log. Then, the monitor attempts to verify the trace log with respect to a given LTL specification using our monitoring algorithm.

We use the *Red Hat OpenStack Platform* servers to generate data. We consider the following *parameters*: (1) number of processes  $|\mathcal{P}|$ , (2) computation duration  $l$ , (3) number of segments  $g$ , (4) event rate per process per second  $r$ , (5) maximum clock skew  $\epsilon$ , (6) number of messages sent per second  $m$ , and (7) LTL formulas under monitoring, in particular, depth of the monitor automaton  $d$ . Our main *metric* to measure is the SMT solving time for each configuration of parameters. Note that in all the plots presented in this section, the time axis is shown in log-scale. When we analyze the effects of one parameter, all the other parameters are held at a relevant constant value. We use a MacBook Pro with Intel i7-7567U(3.5Ghz) processor, 16GB RAM, 512 SSD and Python 3.6.9 interface to the Z3 SMT solver [7]. To evaluate our parallel algorithm, we also use a server with 2x Intel Xeon Platinum 8180 (2.5Ghz) processor, 768GB RAM 112 vcores and python 3.6.9 interface to the Z3 SMT solver [7].

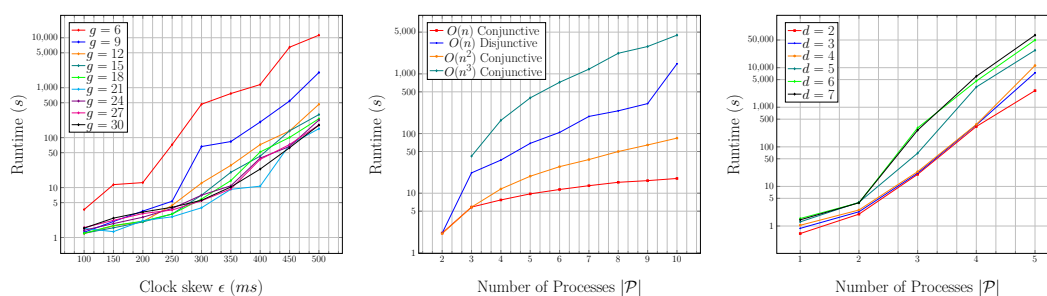
## 4.2 Analysis of Results – Synthetic Experiments

In this set of experiments we attempt to exhaust all the available parameters and metrics discussed earlier. We aim to put all the parameters to test, and examine how they affect the runtime of the verifier. Since the data generated in this case is synthetic and does not depend on any external factors apart from the system configuration, we induce delay after every event in order to uniformly distribute these events throughout the execution of each process, and to achieve different event rates. That is, the events were generated such that they were evenly spread out over the entire simulation. The value of each of the computation events were selected from a uniform distribution over the set  $\Sigma$ .

**Impact of assuming partial synchrony (single core).** Figure 7a shows that with increase in the value of  $\epsilon$ , the runtime increases significantly. This is true for different number of segments. This observation demonstrates that employing HLC and assuming bounded clock skew helps in ordering events and as  $\epsilon$  increases so does the number of concurrent events, and in turn the complexity of verification. Figure 7a also shows that on breaking the computation into smaller segments, the runtime keeps on decreasing for each value of  $\epsilon$ . We will study the impact of segment duration in other experiments as well.

**Impact of predicate structure (single core).** In this experiment (see Fig. 7b), we consider formula  $\Box\varphi$ , and ensure that it remains true throughout the computation duration. This is to ensure that the monitor does not reach a terminal state in the middle of the computation. We consider the following four different predicate structures for  $\varphi$ :

- *O(n) Conjunctive*: In a system of  $n$  processes,  $\varphi$  is a conjunction of  $n$  atomic propositions, each depending on the local state of only one process. Over a set of increasing total number of processes, we observe a linear increase in the runtime. This is somewhat expected, as it is known that monitoring conjunctive predicate is not computationally complex [11].



(a) Different values  $\epsilon$ , LTL formula  $\square p$  ( $d = 2$ ), where  $p$  is a disjunctive predicate and  $|\mathcal{P}| = 2$ . (b) Different predicate structures with  $g = 15$ ,  $d = 2$  and  $\epsilon = 250ms$ . (c) Different formula with  $g = 21$ , predicate structure:  $O(1)$  conjunctive and  $\epsilon = 250ms$ .

■ **Figure 7** Comparison of how the clock skew  $\epsilon$ , structure of predicates and different LTL formulas play a role in monitoring with  $l = 2s$ ,  $r = 10$ , and  $m = 1/s$ .

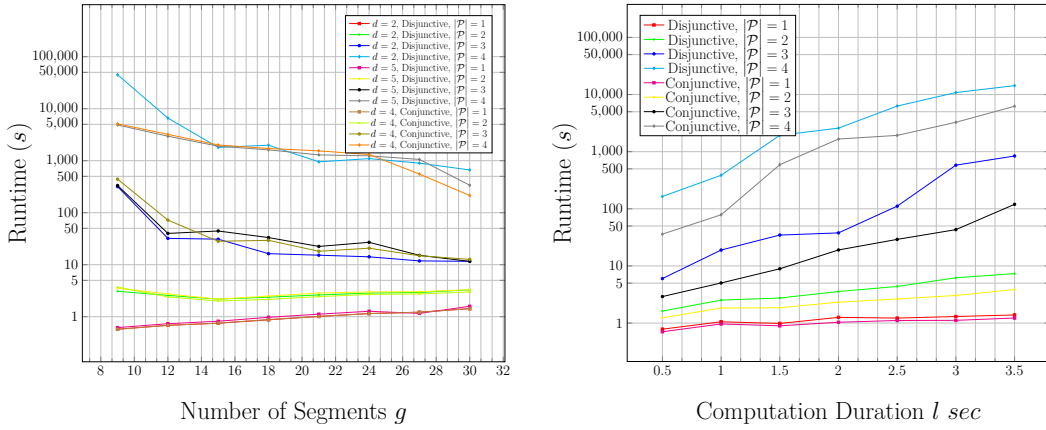
- *$O(n)$  Disjunctive*: Similar to  $O(n)$  conjunctive predicates, here, we have a disjunction of  $n$  atomic propositions. Compared to its conjunctive counterpart, disjunction of propositions requires more time to verify. This follows the theoretical result that monitoring linear predicates is more complex than monitoring regular predicates [9].
- *$O(n^2)$  Conjunctive*: Here,  $\varphi$  is a conjunction of atomic propositions, where each proposition depends on the state of 2 processes, thereby having a total of  $\binom{n}{2}$  predicates. Monitoring such predicates clearly require more time than  $O(n)$  conjunctive predicates, but surprisingly less than  $O(n)$  disjunctive predicates.
- *$O(n^3)$  Conjunctive*: Here, we consider a conjunction of  $\binom{n}{3}$  predicates chosen symbolizing a situation where each predicate is dependent on the state of 3 processes. This case is the most time-consuming structure to monitor.

**Impact of LTL formula (single core).** Given an LTL formula, the depth of the monitor automaton  $d$  is the length of the longest path from the initial to the accept/reject state. In Fig. 7c, we experimented with the following LTL formulas:

$$\begin{array}{ll}
 \varphi_1 = \square(\neg p) & d = 2 \\
 \varphi_2 = \diamond r \rightarrow (\neg p \mathcal{U} r) & d = 3 \\
 \varphi_3 = \square((q \wedge \neg r \wedge \diamond r) \rightarrow (\neg p \mathcal{U} r)) & d = 4 \\
 \varphi_4 = \square((q \wedge \diamond r) \rightarrow (\neg p \mathcal{U} (r \vee (s \wedge \neg p \wedge \mathcal{O}(\neg p \mathcal{U} t)))))) & d = 5 \\
 \varphi_5 = \diamond r \rightarrow (s \wedge \mathcal{O}(\neg r \mathcal{U} t) \rightarrow \mathcal{O}(\neg r \mathcal{U} (t \wedge \diamond p))) \mathcal{U} r & d = 6 \\
 \varphi_6 = \square((q \wedge \diamond r) \rightarrow (p \rightarrow (\neg r \mathcal{U} (s \wedge \neg r \wedge \mathcal{O}(\neg r \mathcal{U} t)))) \mathcal{U} r) & d = 7
 \end{array}$$

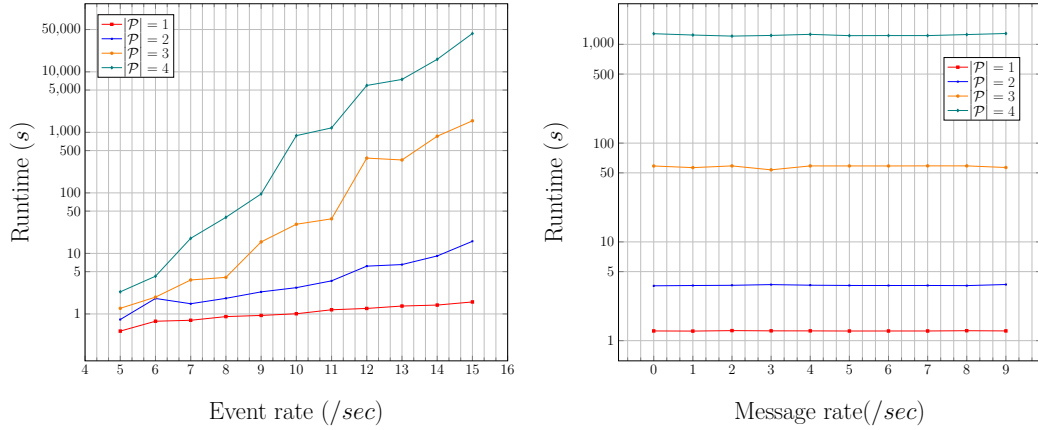
Clearly, deeper monitors incur greater overhead. The predicate structure used is  $O(1)$ , meaning that the predicates are in terms of the state of all processes. Runtime for smaller values of  $d$  are comparable since the overall runtime is dominated by the evaluation of the uninterpreted function  $\rho$  (defined in Section 3). As  $d$  increases, it starts to influence the overall runtime of the verification algorithm.

**Impact of segment count (single core).** As mentioned in Section 3, we anticipate that chopping a distributed computation into smaller segments tackles the intractability of distributed RV, as it may reduce the number of concurrent events. In Fig. 8a, we observe that the runtime keeps on decreasing with increase in the number of segments per computation duration, until it hits a certain level, after which it does not improve any further. This is due


 (a) Impact of segment count with  $l = 2s$ .

 (b) Impact of computation duration with  $g = 21$ .

■ **Figure 8** Impact of segment count and computation duration with  $\epsilon = 250ms$ ,  $O(1)$  predicates,  $r = 10$ ,  $m = 1$ , and formula  $\Box p$ .

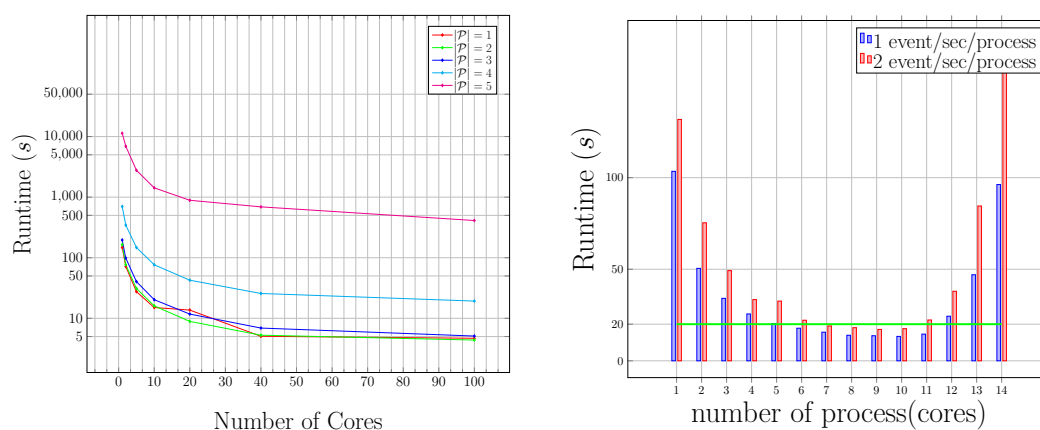

 (a) Impact of event rate with  $m = 1$ .

 (b) Impact of message passing with  $r = 10$ .

■ **Figure 9** Impact of event and message rates with  $l = 2s$ ,  $\epsilon = 250ms$ , and  $g = 21$ , formula  $\Box p$  with  $O(1)$  predicate structure.

to the fact that the total runtime also contains the time required to set up the SMT solver. With increase in the number of segments, the total time required to setup the SMT solver also increases and dominates the speedup. Also, decreasing the segment duration beyond a certain point does not have any effect on the runtime. This is due to the clock skew  $\epsilon$ , which makes each segment start from  $\epsilon$  before. Observe that in Fig. 8a, this result holds for different number of processes, LTL formulas, and conjunctive/disjunctive predicates.

**Impact of computation duration (single core).** The computation duration has a direct effect on the size of  $\mathcal{E}$ , and thus, the number of events in a segment. With a unit increase in the number of events in the SMT formulation, the size of  $2^{\mathcal{E}}$  doubles, increasing the SMT solver search space for  $\rho$ . This makes the runtime in Fig. 8b increase significantly. Observe that in Fig. 8b, this result holds for different number of processes, and conjunctive/disjunctive predicates.



(a) Impact of parallelization with  $l = 20s$ ,  $r = 10$ ,  $\epsilon = 150ms$ ,  $g = 200$ , formula  $\Box p$  with  $O(1)$  predicate structure.

(b) Impact of varying event rate on parallelization for Cassandra.

■ **Figure 10** Impact of parallelization.

**Impact of the event rate (single core).** Until now, the event rate was fixed at 10 events/sec per process, following the latency time obtained in a real network of replicated database (Cassandra), discussed in detail in Section 4.3. Here, we change the event rate and study its effect on the verification runtime. In Fig. 9a, we see increasing the event rate causes the runtime to increase significantly. This result is valid for different number of processes, though for more processes the increase is more dramatic.

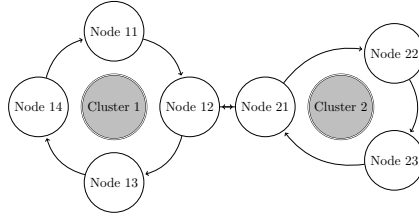
**Impact of the message rate (single core).** Consider a send message event  $e_{\tau,\sigma,\omega}^i$  and its corresponding receive event  $e_{\tau',\sigma',\omega'}^j$ . This results in a  $\rightsquigarrow$ -relation between these two events. Such events are expected to reduce the number of concurrent events and consequently the monitor overhead. However, Fig.9b shows no effect on the monitor run time. This is due to the relatively short  $\epsilon = 250ms$ , which is actually much larger than the maximum clock skew of off-the-shelf protocols such as NTP. In other words, when  $\epsilon$  dominates the impact of event ordering that message passing can achieve. This is another reason to believe that partial synchrony is an effective way to deal with distributed RV. We vary messages sent for inter-process communication, from 0 to 9 with 10events/sec.

**Impact of parallelization.** To demonstrate the drastic increase in performance due to parallelization, we evaluate formula  $\Box p$  on a distributed computation with  $l = 20s$ ,  $r = 10$ ,  $g = 20$ , and  $\epsilon = 150ms$ , while varying the number of cores from 1 to 100, as shown in Fig. 10a. Observe that beyond 40 cores, there is no significant relative change in runtime regardless of the number of processes, as the time required to build the SMT formulation starts dominating the total run time. This graph also shows that parallelization can result in orders of magnitude speedup.

### 4.3 Case Study: Cassandra

Cassandra [15] is a No-SQL database management system. We simulate a system of multiple processes. Each process is responsible for inter-process communication apart from basic database operations (read, write and update). We deployed a system with two data centers (see Fig. 11), where Cluster 1, contains 4 nodes (Nodes 11 – 14) and Cluster 2 contains 3





■ **Figure 11** A network with two Cassandra clusters, Node-12 and Node-21 are the seed nodes of the respective clusters.

nodes (Node 21 – 23). Node 12 and Node 21 are the seed nodes of the respective clusters. Data is replicated in all the nodes in both the clusters. Each of the nodes is part of the *Red Hat OpenStack Platform* with the following configuration: 4 VCPUs, 4GB RAM, Ubuntu 1804, Cassandra 3.11.6, Java 1.8.0\_252, and Python 3.6.9.

We have tested ping time of servers on Google Cloud Platform, Microsoft Azure and Amazon Web Service. The fastest ping was received at  $41ms$ . In a real-life datacenter, networks used to communicate within the nodes usually have a speed on the scale of few Gigabytes per second. Here, we use a private broadband that offers a speed of 100 Megabytes per second. We measure the latency time of our system to be around  $100ms$ . We consider this to be our standard and setup all our experiments based on this assumption.

Processes are capable of reading, writing, and updating all entries of the database. The exact type of the event is selected by a uniform distribution  $(0, 2)$ . Each process selects the available node at run time. In order to prevent deadlocks, no two processes are allowed to connect to the same node at the same time. If there exists no free node at any point of time, it waits for a node to be released and then it continues with the task. Once there is a write or update, the process responsible for the change sends a message to each of the other processes notifying about the change. We assume that a message is read by the receiving process immediately upon receiving. All database operations (i.e. send and receive events) are considered to be separate.

Consistency level in a database dictates the minimum number of replications that needs to perform on an operation in order to consider the operation to be successfully executed. Cassandra recommends that the sum of the read consistency and the write consistency be more than the replication factor for no read or write anomaly in the database. By default, the read and the write consistency level is set to one. For a database with replication factor 3, our goal is to monitor and identify *read/write anomalies* in the database:

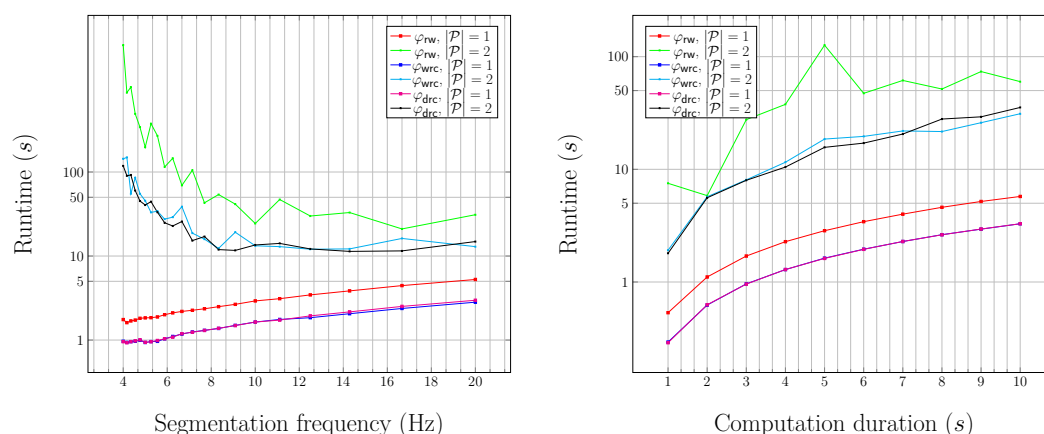
$$\varphi_{rw} = \bigwedge_{i=0}^n \left( write(i) \rightarrow \diamond read(i) \right)$$

where  $n$  is the number of read/write requests.

Since Cassandra does not allow normalization of database, the other two properties we aim to monitor are *write reference check* and *delete reference check*. To give a sense of database normalization, we use a database with two tables:

`Student(id, name)`                      `Enrollment(id, course)`.

We enforce that if there is a write in the `Enrollment` table, it should be led by a write in the `Student` table with the same *id*. The *id* and *name* to be written are a random string of length 8. Likewise, in the case of deletion of some entry from `Student` table, it should be led by deletion of all entries with the same *id* from `Enrollment` table. These enforce that there is



(a) Cassandra: Impact of segmentation frequency for  $l = 5s$ .

(b) Cassandra: Impact of computation duration with segmentation frequency 10Hz.

■ **Figure 12** Experimental results for Cassandra.

no insertion and delete anomaly, and thereby gives a sense of normalization in Cassandra:

$$\varphi_{wrc} = \neg \left( \neg \text{write}(\text{Student.id}) \mathcal{U} \text{write}(\text{Enrollment.id}) \right)$$

$$\varphi_{drc} = \neg \left( \neg \text{delete}(\text{Enrollment.id}) \mathcal{U} \text{delete}(\text{Student.id}) \right)$$

**Extreme load scenario.** Figures 12a and 12b, plot runtime vs segmentation frequency and runtime vs computation duration, respectively for the case where the processes experience full read/write load that network latency allows. Compared to the results plotted for the synthetic experiments, we see a bit of noise in the result. This owes to the fact that in synthetic experiments, the events are uniformly distributed over the entire computation duration, however, in case of Cassandra, the events are not uniform. Database operations like read, write and update take about 100ms of time but sending and receiving of message is relatively faster taking about 20-30ms making the overall event distribution quite non-uniform.

**Moderate load scenario.** In Figure 12a, with event rate  $r = 10$ , we are just about making it even for number of processes as 2 and with a computation duration of 20s. Now, consider Google Sheets API, which allows maximum 500 requests per 100 seconds per project and 100 requests per seconds per user, i.e., 5 events/sec per project and a user can generate 1 event/sec [12] on an average. To see how our algorithm performs in such a scenario, we increase the number of processes and so as the number of monitoring cores and analyze the time taken to verify such a trace log. We plot our findings in Figure 10b. We see that for processes 8, 9 and 10, we get the best results for event rate of both 1 and 2 event(s)/sec/process. We emphasize that the 2 event(s)/sec/process is twice more than what Google Sheets allow to happen. This makes us confident that our algorithm can pave the path for implementation in a real-life setting.

## 5 Related Work

Lattice-theoretic centralized and decentralized online predicate detection in asynchronous distributed systems has been extensively studied in [4, 18]. Extensions of this work to include

temporal operators appear in [20, 19]. The line of work in [4, 18, 20, 19, 22] operates in a fully asynchronous setting. On the contrary in this paper, we leverage a practical assumption and employ an off-the-shelf clock synchronization algorithm to limit the time window of asynchrony. Predicate detection has been shown to be a powerful tool in solving combinatorial optimization problems [10] and our results show that our approach is pretty effective in handling predicate detection (e.g., Fig. 10b). In [24], the authors study the predicate detection problem using SMT solving. Also, knowledge-based monitoring of distributed processes was first studied in [22]. Here, the authors design a method for monitoring safety properties in distributed systems using the past-time linear temporal logic. This approach, however, suffers from producing false negatives.

Runtime monitoring of LTL formulas for synchronous distributed systems has been studied in [8, 6, 5, 1]. This approach has the shortcoming of assuming a global clock across all distributed processes. Predicate detection for asynchronous system has been studied in [23] but the assumption needed to evaluate happen-before relationship is too strong. We utilize HLC which not only is more realistic but also decreases the level of concurrency. Finally, fault-tolerant monitoring, where monitors can crash, has been investigated in [3] for asynchronous and in [13] for synchronized global clock with no clock skew across all distributed processes. In this paper, we use a clock synchronization algorithm which guarantees bounded clock shews. Our solution is also SMT based and to our knowledge this is the first SMT based distributed monitoring algorithm for LTL, which results in better scalability.

## 6 Conclusion and Future Work

In this paper, we focused on runtime verification (RV) of distributed systems. Our SMT-based technique takes as input an LTL formula and a distributed computation (i.e., a collection of communicating processes along with their local events). We employed a partially synchronous model, where a clock synchronization algorithm ensures bounded clock skew among all processes. Such an algorithm significantly limits the impact of full asynchrony and remedies combinatorial explosion of interleavings in a distributed setting. We conducted detailed and rigorous synthetic experiments, as well as a case study on monitoring consistency conditions on Cassandra, a non-SQL replicated database management system used in data centers. Our experiments demonstrate the potential of scalability of our technique to large applications.

As for future work, there are several interesting research directions. Our first step will be to scale up our technique to monitor cloud services with big data. This can be achieved by studying the tradeoff between accuracy and scalability. Another important extension of our work is distributed RV for timed temporal logics. Such expressiveness will allow us to monitor distributed applications that are sensitive to explicit timing constraints. A prominent example of such a setting is in blockchain and cross-chain protocols.

---

## References

- 1 A. Bauer and Y. Falcone. Decentralised LTL monitoring. *Formal Methods in System Design*, 48(1-2):46–93, 2016.
- 2 A. Bauer, M. Leucker, and C. Schallhart. Runtime Verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4):14:1–14:64, 2011.
- 3 B. Bonakdarpour, P. Fraigniaud, S. Rajsbaum, D. A. Rosenblueth, and C. Travers. Decentralized asynchronous crash-resilient runtime verification. In *Proceedings of the 27th International Conference on Concurrency Theory (CONCUR)*, pages 16:1–16:15, 2016.

- 4 H. Chauhan, V. K. Garg, A. Natarajan, and N. Mittal. A distributed abstraction algorithm for online predicate detection. In *Proceedings of the 32nd IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 101–110, 2013.
- 5 C. Colombo and Y. Falcone. Organising LTL monitors over distributed systems with a global clock. *Formal Methods in System Design*, 49(1-2):109–158, 2016.
- 6 L. M. Danielsson and C. Sánchez. Decentralized stream runtime verification. In *Proceedings of the 19th International Conference on Runtime Verification (RV)*, pages 185–201, 2019.
- 7 L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.
- 8 A. El-Hokayem and Y. Falcone. On the monitoring of decentralized specifications: Semantics, properties, analysis, and simulation. *ACM Transactions on Software Engineering Methodologies*, 29(1):1:1–1:57, 2020.
- 9 V. K. Garg. *Elements of distributed computing*. Wiley, 2002.
- 10 V. K. Garg. Predicate detection to solve combinatorial optimization problems. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 235–245. ACM, 2020.
- 11 V. K. Garg and C. Chase. Distributed algorithms for detecting conjunctive predicates. *International Conference on Distributed Computing Systems*, pages 423–430, June 1995.
- 12 Google. Usage limits, sheets api, google developer. <https://developers.google.com/sheets/api/limits>. Accessed: 2020-09-09.
- 13 S. Kazemloo and B. Bonakdarpour. Crash-resilient decentralized synchronous runtime verification. In *Proceedings of the 37th Symposium on Reliable Distributed Systems (SRDS)*, pages 207–212, 2018.
- 14 S. S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone. Logical physical clocks. In *Proceedings of the 18th International Conference on Principles of Distributed Systems*, pages 17–32, 2014.
- 15 Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010. doi:10.1145/1773912.1773922.
- 16 L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.
- 17 D. Mills. Network time protocol version 4: Protocol and algorithms specification. RFC 5905, RFC Editor, June 2010.
- 18 N. Mittal and V. K. Garg. Techniques and applications of computation slicing. *Distributed Computing*, 17(3):251–277, 2005.
- 19 M. Mostafa and B. Bonakdarpour. Decentralized runtime verification of LTL specifications in distributed systems. In *Proceedings of the 29th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 494–503, 2015.
- 20 V. A. Ogale and V. K. Garg. Detecting temporal logic predicates on distributed computations. In *Proceedings of the 21st International Symposium on Distributed Computing (DISC)*, pages 420–434, 2007.
- 21 A. Pnueli. The temporal logic of programs. In *Symposium on Foundations of Computer Science (FOCS)*, pages 46–57, 1977.
- 22 K. Sen, A. Vardhan, G. Agha, and G. Rosu. Efficient decentralized monitoring of safety in distributed systems. In *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, pages 418–427, 2004.
- 23 Scott D. Stoller. Detecting global predicates in distributed systems with clocks. In Marios Mavronicolas and Philippos Tsigas, editors, *Distributed Algorithms*, pages 185–199, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.
- 24 V. T. Valapil, S. Yingchareonthawornchai, S. S. Kulkarni, E. Torng, and M. Demirbas. Monitoring partially synchronous distributed systems using SMT solvers. In *Proceedings of the 17th International Conference on Runtime Verification (RV)*, pages 277–293, 2017.



# Decentralized Runtime Enforcement of Message Sequences in Message-Based Systems

**Mahboubeh Samadi**

University of Tehran, Iran  
mbh.samadi@ut.ac.ir

**Fatemeh Ghassemi**

University of Tehran, Iran  
fghassemi@ut.ac.ir

**Ramtin Khosravi**

University of Tehran, Iran  
r.khosravi@ut.ac.ir

---

## Abstract

In the new generation of message-based systems such as network-based smart systems, distributed components collaborate via asynchronous message passing. In some cases, particular ordering among the messages may lead to violation of the desired properties such as data confidentiality. Due to the absence of a global clock and usage of off-the-shelf components, there is no control over the order of messages at design time. To make such systems safe, we propose a choreography-based runtime enforcement algorithm that given an automata-based specification of unwanted message sequences, prevents certain messages to be sent, and assures that the unwanted sequences are not formed. Our algorithm is fully decentralized in the sense that each component is equipped with a monitor, as opposed to having a centralized monitor. As there is no global clock in message-based systems, the order of messages cannot be determined exactly. In this way, the monitors behave conservatively in the sense that they prevent a message from being sent, even when the sequence may not be formed. We aim to minimize conservative prevention in our algorithm when the message sequence has not been formed. The efficiency and scalability of our algorithm are evaluated in terms of the communication overhead and the blocking duration through simulation.

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

**Keywords and phrases** Asynchronous Message Passing, Choreography-Based, Runtime Enforcement, Runtime Prevention, Message Ordering

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2020.21

## 1 Introduction

The new generation of message-based systems such as network-based smart applications are usually distributed and may consist of off-the-shelf components developed by different vendors. These systems are maintainable and scalable as components collaborate via asynchronous message passing.

Such systems must satisfy the required properties such as data confidentiality, safety, robustness, and security. However, a sequence of messages may lead to the property violation. As an example (inspired by [19]), assume a building that consists of different locations named  $A$ - $E$  where the location  $E$  is restricted and a visitor must enter the restricted location through a legal path (Figure 1). The only legal path to the restricted location is through the consecutive locations  $A$ ,  $C$ , and then  $E$ . Each location is equipped with a smart security camera and a smart door that the visitor must use a smart door to enter the location. The path between different locations is such that if the consecutive locations  $B$  and  $D$  are visited, then the visitor will return to the location  $A$ . If a visitor is entered the restricted location by passing through the consecutive locations  $A$ ,  $B$ , and then  $E$ , it can be inferred that



© Mahboubeh Samadi, Fatemeh Ghassemi, and Ramtin Khosravi;  
licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

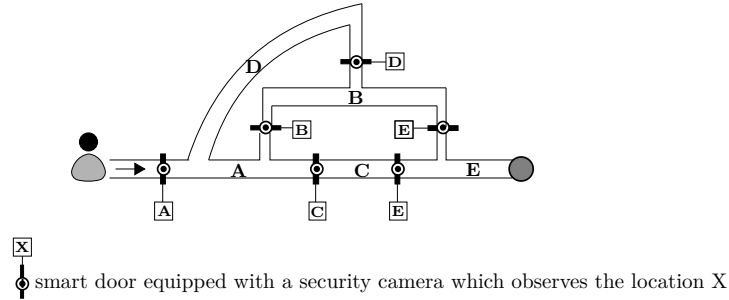
Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 21; pp. 21:1–21:18

Leibniz International Proceedings in Informatics



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

the visitor accesses the restricted location illegally. This illegal access violates the security rules of the building and can be detected by the message sequence  $Open(A,v)$  (the smart door of the location  $A$  has opened by the visitor  $v$ ),  $Open(B,v)$  and  $Open(E,v)$ . There are other examples of sequence-based patterns in the *Complex Event Processing* domain [28, 30]. Furthermore, a *message protocol violation* bug [24] and *linked predicates* [26] are also related to the certain order of communicated messages. The former occurs in an actor-based program [1], as a sample of a message-based program, when the components exchange messages that are not consistent with the intended protocol of the application. The latter defines properties (predicates) on a sequence of events interpreting events to messages.



■ **Figure 1** The locations of a building denoted by A, B, C, D, and E separated by smart doors equipped with security cameras.

As most systems in practice are an integration of various components which may be closed-source and proprietary, the message sequences cannot be inspected statically at design time to guarantee that unwanted sequences never happen at runtime. Runtime enforcement can be used as a verification technique that makes sure such systems satisfy the given properties and correct the execution of the system [23]. In this paper, we focus on the *decentralized* runtime enforcement of properties where each component is equipped with a local monitor. These *decentralized monitors* communicate with each other to prevent the violation of the given property. The given property is violated by the formation of messages sequences, where the sequences obey a specific pattern and specify the particular orderings among sending and receiving messages of distributed components. Upon the occurrence of a message, it may either lead to the sequence formation or cancel the effect of the partially formed sequence. In the previous example, the message sequence  $Open(A,v)$ ,  $Open(B,v)$ ,  $Open(D,v)$ ,  $Open(C,v)$ , and  $Open(E,v)$  does not violate the security rule as the visitor returns to the previous location  $A$  by passing through the consecutive locations  $B$  and  $D$ . Finally, the visitor enters the restricted location  $E$  by passing through the location  $C$ .

Our decentralized runtime enforcement approach (Sect. 3) uses the *choreography* setting [9], where local monitors are organized into a network and collaborate with each other by using a specific protocol. This setting deals with *decentralized specifications* in which each local monitor has access to some parts of the message sequences. The decentralized runtime prevention of messages sequences formation in a message-based system is challenging due to the absence of a global clock and asynchronous message passing. With the absence of a global clock, the order of messages can not be distinguished as components own their local clocks which are not synchronized [34]. With the asynchronous message passing, a component is not synchronized with other components and so it has no information about the status of a sequence formation. In the proposed algorithm (Sect. 4), we will use vector clocks [25] in our messages to detect the *partial* ordering among messages, and then prevent the sequence formation. When monitors cannot detect the *total* order among messages, they



may prevent the sequence formation conservatively in the sense that they prevent a certain message from being sent even if that message does not lead to a sequence formation. We aim to minimize the conservative prevention in our algorithm when the message sequences have not been formed. To prevent a sequence formation, a component may be blocked before sending its message until its monitor makes sure about the effect of that message on the sequence formation. We also aim to prevent the sequence formation by minimizing the number of blocked components and manipulation of messages ordering. To the best of our knowledge, there is no decentralized runtime enforcement of sequence-based properties in message-based systems. We evaluate the performance of our algorithm and show that our algorithm is scalable: with the increase of the complexity of applications or the length of message sequences, the number of monitoring messages and the blocking duration of processes grow linearly (Sect. 5).

## 2 Background

### 2.1 Message-Based Systems

We define a *Message-Based System*  $D = \{P_1, \dots, P_n\}$  as a set of  $n$  processes that communicate via asynchronous message passing and guarantees in-order delivery, i.e., two messages sent directly from one process to another will be delivered and processed in the same order that they are sent. We assume that each process has a unique identifier and a message queue. A process sends messages to a target process using its identifier. Each process takes messages from its queue one by one in FIFO order and invokes a handler regarding the name of the message.

Let  $ID$  be the set of possible identifiers, ranged over by  $x, y$ , and  $z$ . For simplicity, we assume  $ID = \mathbb{N}$  throughout the paper. Let  $MName$  be the set of message names and  $Msg$  be the set of messages communicated among processes ranged over by  $m$ . Each message  $m \in Msg$  has three parts: the sender identifier, the message name, and the receiver identifier, hence  $Msg = ID \times MName \times ID$ . Each process  $P_x$  with the identifier  $x$  is defined by a set of message handlers and state variables where a message handler specifies how the received message must be responded to. The computation of the process  $P_x$  can be abstracted in terms of *events* which are categorized into *internal*, *send*, and *take* events, where an *internal* event changes the state variables of  $P_x$ , the event  $send(P_x, m, P_y)$  occurs when  $P_x$  sends  $m$  to  $P_y$  where  $m \in MName$ , and the event  $take(P_y, m, P_x)$  occurs when  $P_x$  takes  $m$ , which is sent by  $P_y$ , from its queue.

Events in the message-based system can be partially ordered according to the happened-before relation [21] which is implemented by the vector clock. Let a message  $m_i \in Msg$  be a triple of  $(P_x, m_i, P_y)$ . A happened-before relation  $\rightsquigarrow$  defines a causal order among events: (1) within a single message handler, the ordering of events is defined as their execution order which can be determined unambiguously, (2)  $send(P_x, m, P_y) \rightsquigarrow take(P_x, m, P_y)$ , and (3) for events  $e_a, e_b, e_c$ , if  $e_a \rightsquigarrow e_b$  and  $e_b \rightsquigarrow e_c$  then  $e_a \rightsquigarrow e_c$ .

Two events  $e_a$  and  $e_b$  are concurrent and denoted by  $e_a \parallel e_b$  if there is no happened-before relation between them.

### 2.2 Message-Based Property Specification

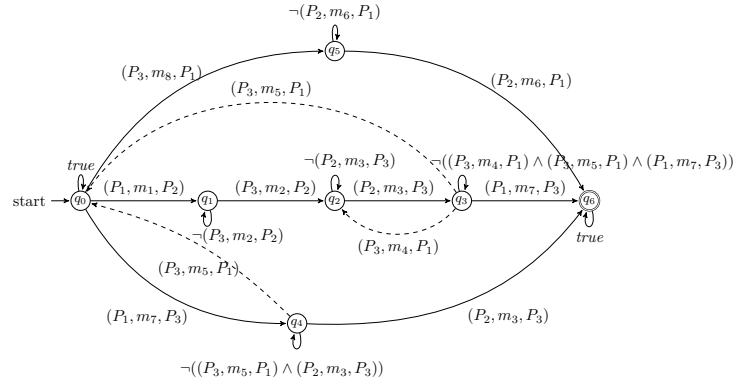
We aim to prevent certain unwanted sequences of send/take events from being formed. For an unwanted sequence, the occurrence of some events contributes to the formation of the sequence, while some other events may cancel the effect of the previous ones. To formalize our message sequences, we use the *sequence automaton* defined in [32] as an extension to

nondeterministic finite automata. In this model, transitions are partitioned into two sets of *forward* and *backward* transitions. Forward transitions, denoted by  $\rightarrow$ , lead to the sequences formation while backward transitions, denoted by  $\dashrightarrow$ , cancel the formation of sequences. Let  $\rightarrow^*$  be the transitive closure of the  $\rightarrow$  relation.

► **Definition 1** (Sequence Automaton [32]). *Given a nondeterministic finite automaton  $(Q, \Sigma, \delta, Q_0, F)$ , the 6-tuple  $(Q, \Sigma, \delta_f, \delta_b, Q_0, F)$  is a sequence automaton (SA), where  $\delta = \delta_f \cup \delta_b$ ,  $\delta_f \cap \delta_b = \emptyset$ , and the transitions specified by  $\delta_f$  (resp.  $\delta_b$ ) are forward (resp. backward) transitions, i.e.,*

- For all simple paths from any initial state  $q_0 \in Q_0$  to any final state  $q_n \in F$  passing through  $q_1 \dots q_{n-1}$ , it holds that  $\forall i < n, q_i \dashrightarrow q_{i+1}$ .
- $q_i \dashrightarrow q_j \Rightarrow q_i \neq q_j \wedge q_j \rightarrow^* q_i$ .

To simplify the explanation, we restrict  $\Sigma$  to *send* events and show  $send(m)$  by  $m$  in our graphical representation of sequence automata.



■ **Figure 2** The sequence automaton  $\mathcal{A}_1$  where the solid edges denote the forward transitions and the dashed edges denote the backward transitions.

The sequence automaton  $\mathcal{A}_1$ , in Figure 2, represents the sequences of *send* events. For instance, this automaton describes that if first the message  $(P_1, m_7, P_3)$  is sent and then the message  $(P_2, m_3, P_3)$  is sent while the message  $(P_3, m_5, P_1)$  is not sent after  $(P_1, m_7, P_3)$  and before  $(P_2, m_3, P_3)$ , then the sequence  $(P_1, m_7, P_3)(P_2, m_3, P_3)$  is formed. If the sequence  $(P_1, m_7, P_3)(P_3, m_5, P_1)(P_2, m_3, P_3)$  is observed, the occurrence of  $(P_3, m_5, P_1)$  has eliminated the effect of the occurrence of  $(P_1, m_7, P_3)$  and so, the occurrence of  $(P_2, m_3, P_3)$  will not form a sequence (as the reaching state  $q_0$  is not a final state). However, a sequence is formed by the occurrence of  $(P_1, m_7, P_3)(P_3, m_5, P_1)(P_1, m_7, P_3)(P_2, m_3, P_3)$ . The self-loop over the state  $q_4$  expresses that between the occurrences of  $(P_1, m_7, P_3)$  and  $(P_2, m_3, P_3)$ , any message except  $(P_3, m_5, P_1)$  and  $(P_2, m_3, P_3)$  can be sent.

When a message  $m$  occurs, a transition like  $(q, m, q')$  may lead to the formation of a sequence from the initial state up to  $q'$ . To form such a sequence, it is necessary that at least a message over one of the preceding transition of  $(q, m, q')$ , like  $t$ , has occurred and no message over the backward transitions has eliminated the effect of  $t$ . The pre-transitions of  $(q, m, q')$  is the set of preceding transitions whose labeled messages can occur before  $m$  in a sequence. The preceding transitions have the same destination as the source state of  $(q, m, q')$ , i.e.,  $q$ .

► **Definition 2** (pre-transition). *For the given sequence automaton  $\mathcal{A}$ , the pre-transitions of the transition  $(q, m, q') \in \delta_f$ , where  $q \neq q'$ , is the set of forward transitions that end in state*

$q$ . Also, the pre-transitions of the transition  $(q, \mathbf{m}, q') \in \delta_b$  is the set of forward transitions that end in state  $q$  and are visited on a path from  $q'$  to  $q$ :

$$\text{preTrns}(\mathcal{A}, (q, \mathbf{m}, q')) = \begin{cases} \{(q'', \mathbf{m}', q) \in \delta_f \mid q \neq q''\} & \text{if } (q, \mathbf{m}, q') \in \delta_f \\ \{(q'', \mathbf{m}', q) \in \delta_f \mid q \neq q'' \wedge q' \rightarrow^* q''\} & \text{if } (q, \mathbf{m}, q') \in \delta_b \end{cases}$$

A backward transition  $(q, \mathbf{m}, q')$  can eliminate the effect of all forward transitions on a path from  $q'$  to  $q$ , when  $\mathbf{m}$  occurs after the occurrence of labeled messages over the sequence of forward transitions on a path from  $q'$  to  $q$ .

► **Definition 3** (vio-transition [32]). *For the given sequence automaton  $\mathcal{A}$ , the vio-transitions of the transition  $(q, \mathbf{m}, q') \in \delta_f$ , where  $q \neq q'$ , is the set of backward transitions that can violate the effect of  $(q, \mathbf{m}, q')$  in a path made up of only forward transitions from the destination to the source of the backward one:*

$$\text{vioTrns}(\mathcal{A}, (q, \mathbf{m}, q')) = \{(q_n, \mathbf{m}', q_0) \mid (q_n, \mathbf{m}', q_0) \in \delta_b \wedge q_0 \rightarrow^* q \wedge q' \rightarrow^* q_n\}$$

For example, in Figure 2,  $\text{preTrns}(\mathcal{A}_1, (q_0, \mathbf{m}_1, q_1)) = \emptyset$ ,  $\text{preTrns}(\mathcal{A}_1, (q_4, \mathbf{m}_3, q_6))$  and  $\text{preTrns}(\mathcal{A}_1, (q_4, \mathbf{m}_5, q_0))$  are equal to  $(q_0, \mathbf{m}_7, q_4)$ . Furthermore, the transition  $(q_4, \mathbf{m}_5, q_0)$  violates the effect of  $(q_0, \mathbf{m}_7, q_4)$  and so  $\text{vioTrns}(\mathcal{A}_1, (q_0, \mathbf{m}_7, q_4)) = \{(q_4, \mathbf{m}_5, q_0)\}$ .

### 3 Choreography-Based Runtime Enforcement Approach

We aim to prevent the formation of unwanted message sequences that are specified by a sequence automaton in a message-based system at runtime. A message sequence  $\mathbf{m}_1 \dots \mathbf{m}_x \dots \mathbf{m}_n$  is formed if we move from the initial state by the message  $\mathbf{m}_1$  and reach a final state by the message  $\mathbf{m}_n$ . To avoid the sequence formation, we equip each process  $P_x$  with a *monitor*  $M_x$ . The local monitors of the processes are organized as a network and communicate with each other to prevent the sequence formation.

To prevent the formation of  $\mathbf{m}_1 \dots \mathbf{m}_{x-1} \mathbf{m}_x \dots \mathbf{m}_n$ , the process  $P_n$  as the sender of  $\mathbf{m}_n$  must make sure that  $\mathbf{m}_1 \dots \mathbf{m}_{n-1}$  has not been formed before sending  $\mathbf{m}_n$ . One possible solution is that its monitor, i.e.,  $M_n$ , communicates with other monitors and asks if all of the messages  $\mathbf{m}_1$  to  $\mathbf{m}_{n-1}$  have been sent. However, this solution imposes a high overhead on the system as there may be many sequences that lead to  $\mathbf{m}_n$ , and so  $M_n$  must ask other monitors about the sending status of many messages. So, we propose a choreography-based prevention approach where monitors have access to some parts of the sequence, detect the sequence formation incrementally, and finally the monitor  $M_n$  informs its process to either send the message  $\mathbf{m}_n$  safely or send an error message. To this end, upon sending a message  $\mathbf{m}_x$ , the sender  $P_x$  informs its monitor  $M_x$ , which in turn asks  $M_{x-1}$  about the formation of  $\mathbf{m}_1 \dots \mathbf{m}_{x-1}$ . The sequence  $\mathbf{m}_1 \dots \mathbf{m}_{x-1} \mathbf{m}_x$  will be formed if  $\mathbf{m}_1 \dots \mathbf{m}_{x-1}$  is formed and  $\mathbf{m}_{x-1} \rightsquigarrow \mathbf{m}_x$ . This way, the communication overhead between the monitors is distributed over time, instead of happening all at the final states.

In the following, first, we explain how monitors have access to some parts of the specification. Then, we demonstrate how monitors must communicate with each other to prevent the formation of unwanted message sequences.

#### 3.1 Choreography-Based Property Specification

In this section, we use the choreography-based specification in [32] where each monitor has its own local property. As we explain in Section 2.2, the message sequences can be specified

## 21:6 Decentralized Runtime Enforcement

by a sequence automaton. To specify the choreography-based specification, the sequence automaton should be broken down into a set of transition tables. Each monitor  $M_x$  maintains a transition table that contains the transitions labeled by the messages that their sender is  $P_x$ . For each transition, the set of its pre-transitions is also stored in the table. Since the effect of a pre-transition may be violated by the occurrence of its vio-transitions, it is necessary to store the set of vio-transitions for each pre-transition in the table too. A transition is uniquely identified in terms of the identifiers of its source/destination states. Self-loops are ignored in the transition tables, as they do not change the state of monitors.

■ **Table 1** The transition table  $\mathcal{T}_{P_1}$ .

transition	final	pre-transition	vio-transition
$(q_0, (P_1, m_1, P_2), q_1)$	$\perp$	$\emptyset$	$\emptyset$
$(q_0, (P_1, m_7, P_3), q_4)$	$\perp$	$\emptyset$	$\emptyset$
$(q_3, (P_1, m_7, P_3), q_6)$	$\top$	$(q_2, @ P_2, q_3)$	$\{(q_3, @ P_3, q_2), (q_3, @ P_3, q_0)\}$

■ **Table 2** The transition table  $\mathcal{T}_{P_2}$ .

transition	final	pre-transition	vio-transition
$(q_2, (P_2, m_3, P_3), q_3)$	$\perp$	$(q_1, @ P_3, q_2)$	$\{(q_3, @ P_3, q_0)\}$
$(q_4, (P_2, m_3, P_3), q_6)$	$\top$	$(q_0, @ P_1, q_4)$	$\{(q_4, @ P_3, q_0)\}$
$(q_5, (P_2, m_6, P_1), q_6)$	$\top$	$(q_0, @ P_3, q_5)$	$\emptyset$

■ **Table 3** The transition table  $\mathcal{T}_{P_3}$ .

transition	final	pre-transition	vio-transition
$(q_1, (P_3, m_2, P_2), q_2)$	$\perp$	$(q_0, @ P_1, q_1)$	$\{(q_3, @ P_3, q_0)\}$
$(q_3, (P_3, m_4, P_1), q_2)$	$\perp$	$(q_2, @ P_2, q_3)$	$\emptyset$
$(q_0, (P_3, m_8, P_1), q_5)$	$\perp$	$\emptyset$	$\emptyset$
$(q_4, (P_3, m_5, P_1), q_0)$	$\perp$	$(q_0, @ P_1, q_4)$	$\emptyset$
$(q_3, (P_3, m_5, P_1), q_0)$	$\perp$	$(q_2, @ P_2, q_3)$	$\emptyset$

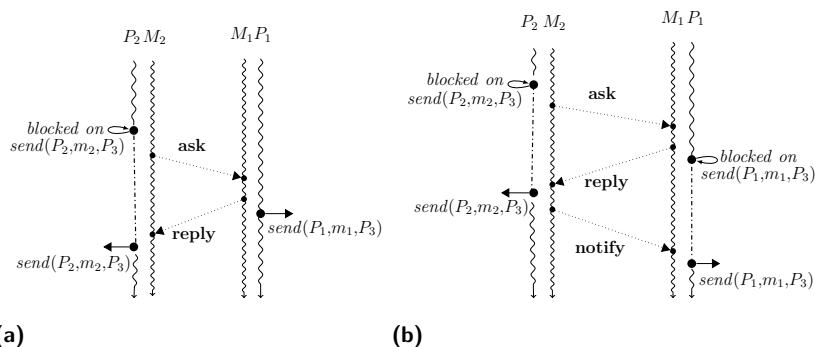
For instance, the automaton in Figure 2 is decomposed into three tables shown in Table 1, 2, and 3. Table 1 is maintained by the monitor of  $P_1$  and contains information of the transitions which the sender of the labeled messages is  $P_1$ . The transition  $(q_0, (P_1, m_1, P_2), q_1)$  does not lead to a final state and it has no pre-transition and so no corresponding vio-transition. So, the first row  $((q_0, (P_1, m_1, P_2), q_1), \perp, \emptyset, \emptyset)$  is included in  $\mathcal{T}_{P_1}$ . The transition  $(q_3, (P_1, m_7, P_3), q_6)$  leads to the final state and has only one pre-transition as  $(q_2, (P_2, m_3, P_3), q_3)$  and two corresponding vio-transitions of  $(q_3, (P_3, m_5, P_1), q_0)$  and  $(q_3, (P_3, m_4, P_1), q_2)$ . The corresponding row of the transition  $(q_3, (P_1, m_7, P_3), q_6)$  in  $\mathcal{T}_{P_1}$  is :

$$((q_3, (P_1, m_7, P_3), q_6), \top, (q_2, @ P_2, q_3), \{(q_3, @ P_3, q_2), (q_3, @ P_3, q_0)\}).$$

### 3.2 Choreography-Based Communication Mechanism

In this section, we demonstrate how monitors communicate with each other to prevent sequences formation. Upon sending a message  $m$  by the process  $P_x$ , the monitor  $M_x$

communicates with other monitors to determine the sequence formation up to  $m$ . In the case that  $m$  is the last message in at least one sequence,  $P_x$  must be blocked before sending  $m$  until  $M_x$  gets information about the partial sequence formation from others and makes sure that sending  $m$  does not complete the sequence formation up to  $m$ . Otherwise,  $P_x$  sends the message  $m$ , and then its monitor tries to detect the sequence formation up to  $m$ .



■ **Figure 3** The monitors collaborate to avoid the sequence formation  $(P_1, m_1, P_3)(P_2, m_2, P_3)$ . The sequence has been formed in (a) as the process  $P_1$  sends the message  $m_1$  immediately before  $M_1$  receives the *notify* message. However, in (b), the sequence has not been formed as the process  $P_1$  sends the message  $m_1$  after  $M_1$  receives the monitoring message *notify*. The dashed part of a thread denotes that the process is blocked until its monitor gets some information from other monitors.

The monitors communicate with each other using monitoring messages. There are three types of monitoring messages called *ask*, *reply*, and *notify*. A monitor sends the monitoring message *ask* to inquire if a message has been sent and receives the response by the monitoring message *reply*.

► **Example 4.** In Figure 3a, the monitors communicate with each other to avoid the sequence formation  $(P_1, m_1, P_3)(P_2, m_2, P_3)$  at runtime. The process  $P_2$  is blocked on the message  $(P_2, m_2, P_3)$  as it is the last message in the sequence. Then,  $M_2$  sends the monitoring message *ask* to  $M_1$  to check if  $m_1$  has been sent. The monitor  $M_1$  responds to  $M_2$  by sending the monitoring message *reply*.

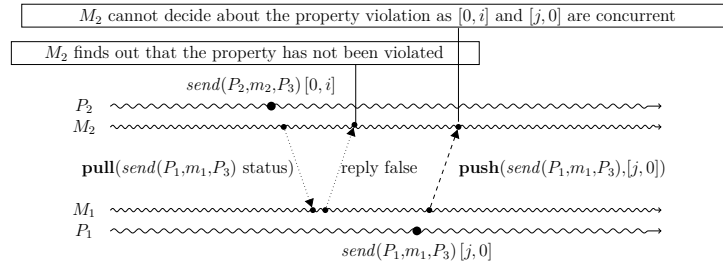
In the case that the process  $P_x$  is blocked on  $m$  until the monitor  $M_x$  makes sure about the completion of the sequence formation,  $M_x$  may receive the response that the inquired message has not been sent. Due to the delay of the network, this response may be received late and meanwhile the inquired message may be sent before receiving this response. So, the process  $P_x$  sends  $m$  and the sequence is formed. To avoid the sequence formation, the inquired message must not be sent by the process of the inquired monitor until the message  $m$  is sent by  $P_x$ , and the monitor  $M_x$  notifies the inquired monitor.

► **Example 5.** In Figure 3a,  $M_1$  responds to  $M_2$  that the message  $(P_1, m_1, P_3)$  has not been sent. However,  $P_1$  sends  $m_1$  immediately after sending this response. In this case, when  $M_2$  receives the response, it finds that  $m_1$  has not been sent and so  $P_2$  can send  $m_2$  safely. But,  $P_2$  sends the message  $m_2$  after sending  $m_1$  and so the sequence has been formed. In Figure 3b, the sequence is not formed as the inquired message  $(P_1, m_1, P_3)$  cannot be sent until  $M_1$  receives the monitoring message *notify* from  $M_2$ .

### 3.2.1 Choreography-Based Communication Strategy

As the message sequence  $m_1 \dots m_{x-1} m_x \dots m_n$  is decentralized between monitors, the monitor  $M_{x-1}$  must inform  $M_x$  the result of the sequence formation  $m_1 \dots m_{x-1}$ . Results can be either pushed into or pulled from a monitor. We use a pulling strategy for collaboration among monitors. With this strategy, monitors find out the order of messages more accurately. We explain the reason through an example.

As we assumed that there is no global clock, processes and monitors append their vector clocks to the events and communicated messages. Consider the property that the event  $send(P_2, m_2, P_3)$  must never occur after the event  $send(P_1, m_1, P_3)$ . Assume that  $P_1$  sends  $m_1$  after  $m_2$  has been sent, but the vector clocks of these messages are concurrent as depicted in Figure 4. With a pushing strategy, the monitor  $M_1$  must inform the monitor  $M_2$  the moment that  $m_1$  has been sent, i.e.,  $[j, 0]$ . When  $P_2$  sends  $m_2$ ,  $M_2$  cannot conclude about the violation of the property as it has not received the moment that  $m_1$  was sent. After pushing the moment of  $m_1$  by  $M_1$ ,  $M_2$  cannot conclude the order among the two events accurately and decide on the property, which is not held, as the vector clocks of the messages are concurrent, i.e.,  $[0, i] \parallel [j, 0]$ . However, with the pulling strategy,  $M_2$  inquires about the sending status of  $m_1$  from  $M_1$  after sending  $m_2$ . If  $P_1$  has not sent  $m_1$  yet, then  $M_1$  responds with a false result. Upon receipt of this response,  $M_2$  can conclude accurately that the property is not violated.



■ **Figure 4** The two communication strategies between the monitors where the pulling strategy is denoted by dotted lines and the pushing strategy is denoted by a dashed line. The vector clock  $[0, i]$ , where  $i > 0$ , denotes that  $P_2$  executes the event  $send(P_2, m_2, P_3)$  as the  $i^{th}$  event while it has no information about the events of  $P_1$ .

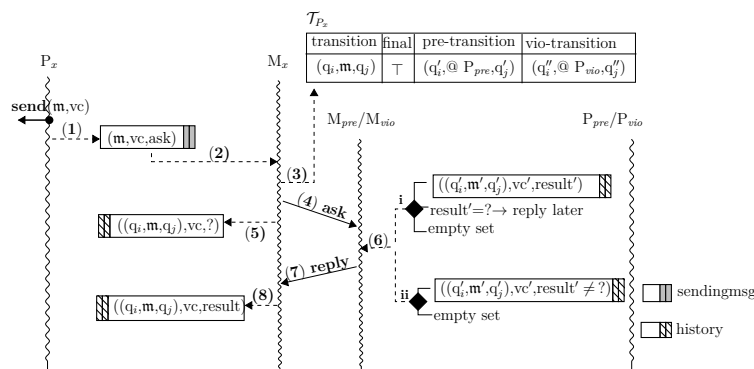
## 4 Choreography-Based Runtime Enforcement Algorithm

In this section, we aim to introduce the choreography-based runtime enforcement algorithm, where the unwanted message sequences are specified on the sequences of  $send$  events.

### 4.1 The Process Environment

The process  $P_x$  maintains the variable  $lastmessages$  which denotes the list of messages labeled on transitions reach to final states. The process is blocked before sending  $m \in lastmessages$  until its monitor makes sure that sending  $m$  does not complete a sequence formation. The process and its monitor also have three shared variables  $sendingmsg_x$ ,  $blockmsg_x$ , and  $waitingmsg_x$ , where:

- $sendingmsg_x$  is a list of triples which consists of a message, which the process is going to send, a vector clock of the process upon sending the message, and the type of a monitoring message which the monitor must send to other monitors.



■ **Figure 5** The algorithm steps taken upon sending the message  $m$  by the process  $P_x$  where  $m$  is not the last message in any sequence.

- $blockmsg_x$  is a pair of a message which  $P_x$  has been blocked on it, and the status of a message to be sent in which it can be either *ok* or *error*.
- $waitingmsg_x$  is the list of messages that must not be sent by  $P_x$  until its monitor receives a notify message as explained in Section 3.2.

We assume that the mutual exclusion of shared variables is ensured by using some well-known mechanisms like semaphore and monitors [20, 17].

## 4.2 The Monitor Environment

The monitor  $M_x$  maintains a transition table  $\mathcal{T}_x$  as described in Section 3.1 to prevent a sequence formation. We call a transition  $t$  of the table  $\mathcal{T}_x$  is *taken* if its labeled message  $m$  has been sent, and a partial sequence up to the transition  $t$  has been formed. A partial sequence up to the transition  $t$  is formed if at least one of its preceding transitions has been taken before, and after that, no violating transition (of those taken preceding transition) has been taken. In the case that  $t$  has no preceding and violating transition, the transition  $t$  is taken when its labeled message has been sent. The time that the transition  $t$  is taken equals the time that  $m$  was sent, and is denoted by a vector clock appended to  $m$ . The monitor  $M_x$  also maintains a variable  $history_x$  which is the list of triples that consists of the transition  $t$  that is taken before, a vector clock of a process upon sending the message  $m$ , and a result of a partial sequence formation up to  $t$ .

## 4.3 The Algorithm Sketch

When the process  $P_x$  wants to send a message  $m$ , there will be two cases depending on whether  $m$  is the last message in at least a sequence. In the following, we explain the behavior of the process and its monitor in the two cases.

### Case 1: $m$ is not the last message in any sequence

If  $m$  is not the last message in any sequence, i.e.,  $m \notin lastmessages$ , the process  $P_x$  sends the message and appends the triple  $(m, vc, ask)$  to the end of  $sendingmsg_x$ . The monitor  $M_x$  takes a message from  $sendingmsg_x$ . If the type of message is *ask*, it inspects if any transition of  $\mathcal{T}_x$  can be taken. Then,  $M_x$  finds those rows of  $\mathcal{T}_x$  whose labeled message on its transition equals  $m$ . For each row,  $M_x$  inquires about the taken status of the pre-transition and vio-transitions in the row by sending appropriate monitoring messages to the monitors



corresponding to the sender of the messages over these transitions. Additional information is appended to the monitoring messages including the vector clock of the sending event of  $\mathbf{m}$ , the inspected transition of  $\mathcal{T}_x$  labeled by  $\mathbf{m}$ , called  $t$ , the blocked status of the process  $P_x$  on  $\mathbf{m}$ , the type of the monitoring message, and the inquired transitions. Then,  $M_x$  adds the temporary record  $(t, vc, ?)$  to its history. The triple  $(t, vc, result)$  expresses that the taken status of the transition  $t$  that its labeled message was sent at the moment  $vc$ , is either under inspection or defined. The former case is indicated by the result value of “?” while the latter is indicated by the result values of  $Frm$  or  $Frm_p$  which are explained later. Adding the record  $(t, vc, ?)$  is helpful when another monitor inquires  $M_x$  about the taken status of the transition  $t$ . In such cases, the monitor  $M_x$  must postpone its response to the inquiry until the result of the transition  $t$  be defined. Figure 5 shows the steps of the algorithm in this case and (1) – (5) denotes the steps explained so far.

► **Example 6.** In Figure 2, when  $P_2$  sends the message  $m_3$ , it appends the triple  $(\mathbf{m}_3, vc, ask)$  to the end of  $sendmsg_2$ . Upon taking this triple from  $sendmsg_2$ , the monitor  $M_2$  checks the transition table  $\mathcal{T}_{P_2}$  (Table 2) and finds two transitions labeled by  $(P_2, m_3, P_3)$ . For instance, as the transition  $(q_4, (P_2, m_3, P_3), q_6)$  has one pre-transition and one vio-transition,  $M_2$  prepares two monitoring messages to inquire about the taken status of the pre-transition  $(q_0, @P_1, q_4)$  from  $M_1$  and the vio-transition  $(q_4, @P_3, q_0)$  from  $M_3$ . Then,  $M_2$  adds the triple  $((q_4, (P_2, m_3, P_3), q_6), vc, ?)$  to its history.

Upon receiving the monitoring message  $ask$  by  $M_y$ , there are two cases according to the blocked status of  $P_y$  which is the sender of a message  $\mathbf{m}'$  labeled on the inquired transition:

(1) *The process  $P_y$  is not blocked on  $\mathbf{m}'$ :* In this case, if  $M_y$  has either an unknown result “?” in  $history_y$  corresponded to the inquired transition or an unhandled message in  $sendmsg_y$  corresponded to  $\mathbf{m}'$  in which  $send(\mathbf{m}') \rightsquigarrow send(\mathbf{m})$ , then it must postpone responding to the monitoring message. Otherwise, if  $M_y$  finds a record with a defined result value about the inquired transition, it infers that the transition has been previously taken. If so,  $M_y$  attaches the corresponding information found in its history to its response monitoring message. If there is no record with a defined result value about the inquired transition, it attaches an empty set to the monitoring message (6i). Then,  $M_y$  communicates with  $M_x$  by sending the monitoring message *reply*.

► **Example 7.** In Figure 2, suppose that the monitor  $M_2$  inspects the taken status of the transition  $(q_4, (P_2, m_3, P_3), q_6)$ , and inquires about the taken status of  $(q_0, @P_1, q_4)$  from  $M_1$ . If  $M_1$  finds any record  $((q_0, \mathbf{m}', q_4), vc', ?)$  in  $history_1$ , where  $send(\mathbf{m}') \rightsquigarrow send(P_2, m_3, P_3)$ , then it postpones responding to this monitoring message until the result value “?” be defined. Otherwise, it attaches the found records to the monitoring message and send to  $M_2$ .

(2) *The process  $P_y$  is blocked on  $\mathbf{m}'$ :* The monitor  $M_y$  checks  $history_y$  to investigate whether  $\mathbf{m}'$  has been previously sent and there is any record with a defined result value corresponding to the inquired transition. If such a record with a defined result value is found,  $M_y$  attaches the found information to its response monitoring message. Otherwise, it attaches an empty set to the monitoring message (6ii). Then,  $M_y$  communicates with  $M_x$  by sending the monitoring message *reply*. In this case, there may be a record corresponding to the inquired transition with the unknown result “?” in  $history_y$ . However, a defined value of this result does not affect on the sequence formation as  $\mathbf{m}'$  has not been sent yet, and the inspected message  $\mathbf{m}$  has been sent by  $P_x$ . So,  $M_y$  can send the records with a defined result value to  $M_x$  irrespective of the records with an unknown value about the inquired transition.



to either  $(\mathbf{m}, ok)$  or  $(\mathbf{m}, error)$ . The pair  $(\mathbf{m}, ok)$  denotes that no sequence will be formed by sending  $\mathbf{m}$ . In this case,  $P_x$  can continue its execution and send the message safely. The pair  $(\mathbf{m}, error)$  denotes that a sequence up to the last message  $\mathbf{m}$  has been formed and so sending  $\mathbf{m}$  leads to a complete sequence formation. The monitor  $M_x$  behaves similarly to the previous case upon taking a message from  $sendmsg_x$  (2) – (4). Then,  $M_x$  adds the temporary record  $(t, -, ?)$  to  $history_x$  (5).

Upon receiving the monitoring message by  $M_y$ , there are two cases based on the blocked status of  $P_y$  which is the sender of the message labeled on the inquired transition, i.e.,  $\mathbf{m}'$ :

(1) *The process  $P_y$  is not blocked on  $\mathbf{m}'$* : The monitor  $M_y$  behaves similarly to the first item of the previous case, except that it must also add  $\mathbf{m}'$  to  $waitingmsg_y$  if it finds no record with a defined result value about the inquired transition (6i).

(2) *The process  $P_y$  is blocked on  $\mathbf{m}'$* : As the process  $P_y$  is blocked on  $\mathbf{m}'$ , there will be a record  $((q, \mathbf{m}', q'), -, ?)$  in  $history_y$ . The monitor  $M_y$  must postpone responding to this monitoring message as it does not know whether a sequence up to  $\mathbf{m}'$  is formed (6ii).

The monitor  $M_x$  behaves similarly to the previous case upon receiving all responses from other monitors. If a sequence up to  $\mathbf{m}$  is formed, then  $M_x$  updates  $blockmsg_x$  to  $(\mathbf{m}, error)$  to inform  $P_x$  that sending  $\mathbf{m}$  leads to a complete sequence formation. Otherwise, it updates  $blockmsg_x$  to  $(\mathbf{m}, ok)$  to inform  $P_x$  that  $\mathbf{m}$  can be sent safely (7) – (9).

The process  $P_x$  either sends  $\mathbf{m}$  or sends an error message regarding the status of the message in  $blockmsg_x$ , and appends the triple  $(\mathbf{m}, vc, notify)$  to the end of  $sendmsg_x$  (10) – (11). The monitor  $M_x$  takes the triple with the message type  $notify$  from  $sendmsg_x$  and sends the corresponding monitoring message (12) – (13). If  $P_x$  sends  $\mathbf{m}$  by the vector clock  $vc$ ,  $M_x$  also updates the vector clock of the transition labeled by  $\mathbf{m}$  in  $history_x$  from “–” to  $vc$ . Finally, the monitor  $M_y$  which receives the notify message from  $M_x$ , removes the message labeled on the inquired transition from  $waitingmsg_y$  (14).

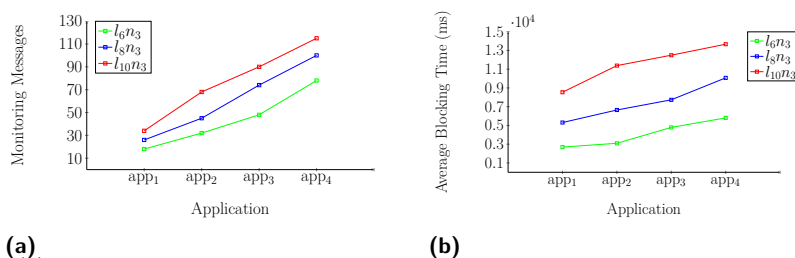
#### 4.4 Discussion

We have assumed that processes and their monitors behave honestly and do not suffer from any failures or byzantine behavior [14]. If a monitor fails or a process fails before updating the shared variable, the algorithm will not be sound due to the loss or the incomplete information of the monitor. In cases that processes tamper with events or behave maliciously, they may not inform their monitors upon the occurrence of *send/take* events. Hence, monitors conclude wrongly and the algorithm will not be sound again. The proposed algorithm can be implemented in the execution framework of message-based systems. For instance, the send function of the open source Akka library [2] or the control layer of Theater [8] which regulates the message scheduling and dispatching can be modified to incorporate our enforcement algorithm. On the other hand, the given specification for the unwanted message sequences may lead our algorithm to reach a communication deadlock [31] among the monitors. For instance, suppose that we aim to prevent the formation of the message sequences  $\mathbf{m}_1\mathbf{m}_2$  and  $\mathbf{m}_2\mathbf{m}_1$ . According to our algorithm, the process  $P_2$  is blocked before sending the message  $\mathbf{m}_2$  and then  $M_2$  asks  $M_1$  if  $\mathbf{m}_1$  has been sent. The process  $P_1$  also may be blocked before sending  $\mathbf{m}_1$  and then  $M_1$  inquires  $M_2$  if  $\mathbf{m}_2$  has been sent. So, both processes  $P_1$  and  $P_2$  will be blocked as their monitors cannot determine the sequence formation up to their blocked messages. However, our algorithm works correctly for sequences without such dependencies. The proof sketch of our algorithm is given in Appendix A. We are working on an extended version of our algorithm to detect and resolve communication deadlocks as a future work.

## 5 Evaluation and Experimental Results

In this section, we present the results of a set of experiments to evaluate our runtime enforcement algorithm. We investigate the effect of different parameters on the efficiency of the algorithm including the number of processes, the maximum number of message handlers of processes, the maximum message communication chain between processes, and the length of the message sequences. The maximum message communication chain denotes the maximum number of processes in a chain of message handlers that send messages to each other. We develop a test case generator <sup>1</sup> which produces message-based applications with different parameters and a set of message sequences according to the generated application. Applications are generated in terms of a simple actor-based language [1]. We also develop a simulator <sup>2</sup> which simulates the execution of each application and our prevention algorithm, and then measures the communication overhead of our algorithm. The simulator tools assume a random network delay and our simulator delivers messages after this delay. We perform all the experiment on a single machine with a dual core processor (Intel i5-520M 2.4GHz) with 4 GB memory.

To evaluate the scalability and the monitoring communication overhead of our algorithm, we generate four applications with 3, 6, 9, and 12 processes, where each process has maximum five message handlers, and the maximum message communication chain in each application is 4, 5, 6, and 7, respectively.



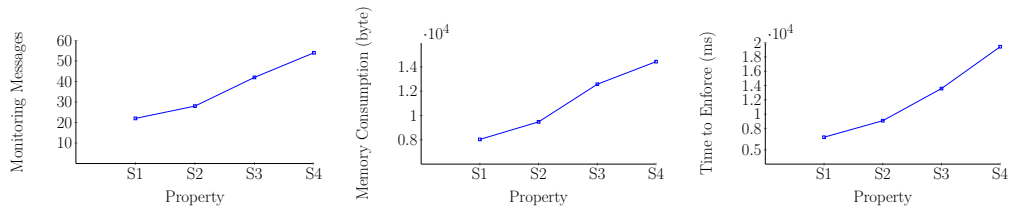
**Figure 7** (a) The average number of monitoring messages in different applications regarding three sequences (denoted by  $n_3$ ) with different lengths of six (denoted by  $l_6$ ), eight, and ten, (b) The average time that processes are blocked regarding the same message sequences of (a).

**Scalability:** To show that our algorithm is scalable in terms of the average number of monitoring messages and the average blocking time of processes, we run each experiment ten times. The average number of monitoring messages for each application is shown in Figure 7a where the given properties are three message sequences with the length of six, eight, and ten. The number of times that a monitor inquires others because of a message  $m$  (occurring in sequences) depends on the number of times that the message  $m$  has occurred at runtime. To make our experiments fair, we enforce the restriction that each message can appear in at most two communication chains. In this case, each constituent message of the sequence can occur at most two times. Our results show that as the length of message communication chain increases, the number of monitoring messages grows linearly for complex applications.

We also evaluate the average time that the processes of each application are blocked. In the proposed algorithm, only the senders of the last messages in the sequences are blocked. Figure 7b shows that the average blocking time of processes grows linearly for complex application to prevent the formation of three message sequences with the length of six, eight, and ten.

<sup>1</sup> Available at <https://gitlab.com/vmoh.ir/rebeca-generator>, Accessed: 2020-11-04

<sup>2</sup> Available at <https://gitlab.com/mSamadi/enforcement>, Accessed: 2020-11-04



**Figure 8** (a) the average number of monitoring messages, (b) the average of memory consumption, (c) the average time for preventing the message sequence formation for different properties where  $S_i$  denotes three sequences with the length of  $i + 4$ .

### Monitoring Communication Overhead:

We evaluate the average number of monitoring messages, the average of memory consumption of the monitors, and the average time to enforce a property for the application with nine processes. As illustrated in Figure 8a and Figure 8b, the average number of monitoring messages and the average memory consumption of the monitors grows linearly as the length of sequences increases. To measure the average time to enforce the property, we measure the average time that the monitors are waited for receiving the responses from other monitors, plus the total time that a process is blocked until its monitor informs it to send a message. It is shown in Figure 8c that as the length of the sequence increases, the monitors involve in more collaborations and hence, more time to gather all responses from other monitors.

## 6 Related Work

Several *centralized* monitoring algorithms [34, 4, 9] and *decentralized* ones [32, 27, 3] have been proposed to *detect* the property violation in distributed systems at runtime. Among the centralized runtime enforcement approaches which aim to *avoid* the property violation, we can mention [33] which introduces security automata to specify security properties. Using this model, the execution of the program is stopped if a sequence of events does not satisfy the desired property. Using the edit automaton [22], the execution of the program can be corrected by suppressing or inserting a new event. This automaton assumes that monitors can predetermine the results of events without executing them. In [10, 23], an enforcement model is presented for the cases that the results of events are not predetermined. In the presented model, for every event generated by the program, the underlying executing system returns a result to the target program. The predictive runtime enforcement [29] deals with systems that are not entirely black-box, and there is some knowledge about their behavior. The knowledge allows to output some events immediately, and the system is not blocked until more events are observed. The timed properties are enforced, in [12] at runtime. Furthermore, in [6], an enforcement approach for the reactive systems is presented where the output should be corrected only if necessary, as little as possible, and without delay. In addition to these work, [11, 7] deals with the runtime enforcement of component-based systems, where systems are modeled within the BIP framework [5]. In this approach, monitors are synchronized with their components. However, the proposed algorithm is decentralized and monitors collaborate to prevent the unwanted sequences formation.

The existing approaches in the domain of runtime enforcement are categorized in [13], and decentralized runtime enforcement is considered as an open challenge in distributed systems. We can address [15] as a decentralized enforcement approach in which a framework, called service automata, is specified in Hoare’s CSP language [18]. This framework considers networks of service automata that are not fully connected. Each service automaton

synchronizes with the system on the critical events. This automaton controls the execution of a program and communicates with other service automata to decide whether a property is satisfied. However, in our choreography-based approach, the monitors are fully connected and a monitor can communicate with others monitors directly and so fewer monitoring messages are transmitted in the network. In addition, there is no synchronization among processes and their monitors, and the monitors take advantage of the specific communication mechanism (Sect. 3.2) to prevent the scenario of sequence formation given in Figure 3. The work of [16] is considered as a decentralized enforcement approach in the domain of business processes where a document must follow a specific workflow. It uses the notion of migration strategy [9] where the document is transmitted among different parties. The document carries fragments of its history, and is protected from tampering using hashing and encryption. Here, the workflow as a specification is shared among different parties as opposed to our method.

## 7 Conclusion and Future work

We addressed the choreography-based runtime prevention of message sequences formation in systems where distributed processes communicate via asynchronous message passing. We have assumed that there is no global clock and the network may postpone delivery of messages. Our proposed algorithm is fully decentralized in the sense that each process is equipped with a monitor which has partial access to some parts of the property specification. Monitors cannot identify the total ordering among messages using the vector clock and hence, may prevent a sequence formation conservatively. We developed a simulator to evaluate the effect of different application and the length of the message sequences on various factors, including the number of monitoring messages, memory consumption of the monitors, and the time to prevent the sequence formation. Our experimental results show that with the increase of the complexity of application or the length of message sequences, the number of monitoring messages, memory consumption, and the time to prevent the sequence formation grows linearly. We are going to resolve the possible communication deadlock based on the given message sequences in the future and integrate our algorithm with the AKKA library.

---

### References

- 1 Gul. Agha. *ACTORS - a model of concurrent computation in distributed systems*. MIT Press series in artificial intelligence, 1990.
- 2 Akka. <https://akka.io>. Accessed: 2020-09-09.
- 3 Bavid Basin, Felix Klaedtke, and Eugen Zălinescu. Runtime verification of temporal properties over out-of-order data streams. In *Proc. CAV*. Springer, 2017.
- 4 Andreas Bauer and Yliès Falcone. Decentralized LTL monitoring. In *Proc. FM*. Springer, 2012.
- 5 Simon Bliudze and Joseph Sifakis. The algebra of connectors—structuring interaction in bip. *Journal of TC. IEEE*, 57(10):1–16, 2008.
- 6 Roderick Bloem, Bettina Könighofer, Robert Könighofer, and Chao Wang. Shield synthesis: runtime enforcement for reactive systems. In *Proc. TACAS*. Springer, 2015.
- 7 Hadil Charafeddine, Khalil El-Harake, Yliès Falcone, and Mohamad Jaber. Runtime enforcement for component-based systems. In *Proc. SAC*. ACM, 2015.
- 8 Franco Cicirelli, Libero Nigro, and Paolo Sciammarella. Model continuity in cyber-physical systems: a control-centered methodology based on agents. *Journal of Simul Model Pract Theory. Elsevier*, 83:93–107, 2018.
- 9 Christian Colombo and Yliès Falcone. Organising LTL monitors over distributed systems with a global clock. *Journal of FMSD. Springer*, 42(1):109–158, 2016.

- 10 Egor Dolzhenko, Jay Ligatti, and Srikar Reddy. Modeling runtime enforcement with mandatory results automata. *JIS. Springer*, 14(2):47–60, 2015.
- 11 Yliès Falcone and Mohamad Jaber. Fully automated runtime enforcement of component-based systems with formal and sound recovery. *Journal of STTT. Springer*, 19(3):341–365, 2017.
- 12 Yliès Falcone, Thierry Jéron, Hervé Marchand, and Srinivas Pinisetty. Runtime enforcement of regular timed properties by suppressing and delaying events. *Journal of SCP. Elsevier*, 123(1):2–41, 2016.
- 13 Yliès Falcone, Leonardo Mariani, Antoine Rollet, and Saikat Saha. Runtime failure prevention and reaction. In *Lectures on Runtime Verification*. Springer, 2018.
- 14 Yliès Falcone, Robert Shostak, and Marshall Pease. The byzantine generals problem. *Journal of TOPLAS. ACM*, 4(3):382–401, 1982.
- 15 Richard Gay, Heiko Mantel, and Barbara Sprick. Service automata. In *Proc. FAST*. Springer, 2011.
- 16 Sylvain Hall, Raphaël Khoury, Quentin Betti, Antoine El-Hokayem, and Yliès Falcone. Decentralized enforcement of document lifecycle constraints. *JIS. ACM*, 74(2), 2018.
- 17 Brinch Hansen. *The Origin of Concurrent Programming*. Springer-Verlag, 2002.
- 18 C.A.R. Hoare. *Communicating sequential processes*. Prentice-Hall International series in computer science, 1985.
- 19 Ilya Kolchinsky and Assaf Schuster. Efficient adaptive detection of complex event patterns. In *Proc. VLDB*. ACM, 2018.
- 20 Ajay D Kshemkalyani and Mukesh Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2011.
- 21 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *JCM. ACM*, 21(7):558–565, 1978.
- 22 Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: Enforcement mechanisms for run-time security policies. *JIS. Springer*, 4(1):2–16, 2005.
- 23 Jay Ligatti and Srikar Reddy. A theory of runtime enforcement, with results. In *Proc. ESORICS*. Springer, 2010.
- 24 Carmen. T Lopez, Stefan Marr, Elisa Gonzalez, and Hanspeter Mössenböck. *A Study of Concurrency Bugs and Advanced Development Support for Actor-based Programs, Programming with Actors*. Programming with Actors, 2018.
- 25 Friedemann Mattern. *Virtual Time and Global States of Distributed Systems, Parallel and Distributed Algorithms*. North-Holland Press, 1988.
- 26 Barton. P Miller and J. D Choi. Breakpoints and halting in distributed programs. In *Proc. ICDCS*. IEEE, 1988.
- 27 Menna Mostafa and borzoo Bonakdarpour. Decentralized runtime verification of ltl specifications in distributed systems. In *Proc. IPDPS*. IEEE, 2015.
- 28 Saravana.M Palanisamy, Muhammad.A Tariq Frank Dürr, and Kurt Rothermel. Preserving privacy and quality of service in complex event processing through event reordering. In *Proc. DEBS*. ACM, 2018.
- 29 Srinivas Pinisetty, Viorel Preoteasa, Stavros Tripakis, Thierry Jéron, Yliès Falcone, and Hervé Marchand. Predictive runtime enforcement. *Journal of FMSD. Springer*, 51(3):154–199, 2017.
- 30 Yingmei Qi, Lei Cao, Medhabi Ray, and Elke A. Rundensteiner. Complex event analytics: online aggregation of stream sequence patterns. In *Proc. SIGMOD*. ACM, 2014.
- 31 Panos Rondogiannis, Georgios Pavlides, and A. Levy. Distributed algorithm for communication deadlock detection. *Journal of IST. Elsevier*, 33(7):483–488, 1991.
- 32 Mahboubeh Samadi, Fatemeh Ghassemi, and Ramtin Khosravi. Choreography-based runtime verification of message sequences in distributed message-based systems. available in <http://rebeca-lang.org/assets/papers/archive/rv.pdf>.
- 33 Fred. B Schneider. Enforceable security policies. *Journal of TISSEC. ACM*, 3(1):30–50, 2000.



- 34 César Sánchez, Gerardo Schneider, Wolfgang Ahrendt, and et al. A survey of challenges for runtime verification from advanced application domains (beyond software). *Journal of FMSD*. Springer, 54(3):273–335, 2018.

## A Soundness of The Algorithm

We aim to prove that the proposed algorithm is *sound* meaning that the output of a message-based system is correct and no given message sequences will be formed at runtime.

► **Lemma 9.** *For any sequence  $\mathbf{m}_1 \dots \mathbf{m}_{n-1} \mathbf{m}_n \in \mathcal{L}(\mathcal{A})$ , the corresponding monitor of  $\mathbf{m}_{n-1}$  declares the formation of  $\mathbf{m}_1 \dots \mathbf{m}_{n-1}$  correctly.*

**Proof.** It is trivial that if  $\omega = \mathbf{m}_1 \dots \mathbf{m}_{n-1} \mathbf{m}_n \in \mathcal{L}(\mathcal{A})$ , then there exists at least a sub-sequence  $\mathbf{m}_1^i \mathbf{m}_2^j \mathbf{m}_3^w \dots \mathbf{m}_l^h \dots \mathbf{m}_n^k$ , called  $\hat{\omega}$ , where for the message  $\mathbf{m}_l^h$ ,  $h$  is the index of the message in  $\omega$  and  $l$  is the index of the message in  $\hat{\omega}$ , i. e.,  $0 \leq l \leq n$ . For each pair of  $\mathbf{m}_{i'}^{i'} \mathbf{m}_{j'+1}^{j'}$  of  $\hat{\omega}$ , there is no message  $\mathbf{m}_o$  in  $\omega$ , where  $i' < o < j'$ , that cancel the effect of  $\mathbf{m}_{i'}^{i'}$ . In other words, the messages of  $\hat{\omega}$  comprise of only forward transitions from the initial state of  $\mathcal{A}$  to the final state, and for each pair of  $\mathbf{m}_{i'}^{i'} \mathbf{m}_{j'+1}^{j'}$ ,  $\mathbf{m}_{i'}^{i'}$  occurs as the label of a pre-transition of the transition carrying  $\mathbf{m}_{j'+1}^{j'}$ . We show that the corresponding monitor of  $\mathbf{m}_{n-1}$ , declares the formation of  $\mathbf{m}_1 \dots \mathbf{m}_{n-1}$  correctly.

The message  $\mathbf{m}_i$  has occurred before the message  $\mathbf{m}_j$ , denoted by  $\mathbf{m}_i \rightarrow \mathbf{m}_j$ , if and only if  $\neg(vc(\mathbf{m}_j) < vc(\mathbf{m}_i))$ . So, it can be concluded that in  $\hat{\omega}$ , either  $\mathbf{m}_1^i$  has happened before  $\mathbf{m}_2^j$  or  $\mathbf{m}_1^i$  is concurrent with  $\mathbf{m}_2^j$  i.e.,  $\mathbf{m}_2^j \not\prec \mathbf{m}_1^i$  in short. By running our algorithm, the monitor of  $\mathbf{m}_2^j$ , namely  $M_2$ , checks the taken status of its pre-transitions and the vio-transitions of the pre-transitions ((3) in Figure 5). So, a transition labeled by  $\mathbf{m}_1^i$ , called  $t$ , and its corresponding vio-transitions are investigated. If a message belonging to the vio-transitions of  $t$ , called  $\mathbf{m}_v$ , has occurred in  $\omega$ , it must have occurred before  $\mathbf{m}_1^i$ , where  $v < i$ , in  $\omega$  due to our condition on the sub-sequence. Two cases can be distinguished: either the message  $\mathbf{m}_v$  has happened before  $\mathbf{m}_1^i$  or  $\mathbf{m}_v$  is concurrent with  $\mathbf{m}_1^i$ . In the first case, where the vector clock of  $\mathbf{m}_v$  is less than the vector clock of  $\mathbf{m}_1^i$ , the effect of  $\mathbf{m}_i$  has not been canceled by  $\mathbf{m}_v$ . So,  $M_2$  checks the vector clocks of  $\mathbf{m}_1^i$  and  $\mathbf{m}_2^j$ . If  $\mathbf{m}_1^i$  has happened before  $\mathbf{m}_2^j$ ,  $M_2$  concludes that the sequence  $\mathbf{m}_1^i \mathbf{m}_2^j$  has been formed so far and stores the “*Frm*” result in its history. If there is no relation between the vector clocks of  $\mathbf{m}_1^i$  and  $\mathbf{m}_2^j$ ,  $M_2$  behaves conservatively and stores the result value “*Frm<sub>p</sub>*” in its history ((8) in Figure 5). In the second case, where the vector clock of  $\mathbf{m}_1^i$  is concurrent with  $\mathbf{m}_v$ ,  $M_2$  does not know whether  $\mathbf{m}_v$  has cancel the effect of  $\mathbf{m}_1^i$  and so it adds the result value “*Frm<sub>p</sub>*” to its history since  $\mathbf{m}_2^j \not\prec \mathbf{m}_1^i$ .

Up to here, the result with “*Frm*” or “*Frm<sub>p</sub>*” value has been correctly inserted into the  $M_2$ ’s history. With the same discussion, we select  $\mathbf{m}_3^w$  of  $\hat{\omega}$  and assume that the message  $\mathbf{m}_{v'}$ , which cancels the effect of  $\mathbf{m}_2^j$ , has occurred and due to our condition on the sub-sequence, it must have occurred before  $\mathbf{m}_2^j$  in  $\omega$ . So, the message  $\mathbf{m}_2^j$  has happened before  $\mathbf{m}_3^w$  or  $\mathbf{m}_2^j$  is concurrent with  $\mathbf{m}_3^w$ . By applying our algorithm, the monitor of  $\mathbf{m}_3^w$ , namely  $M_3$ , inquiries about  $\mathbf{m}_2^j$  and  $\mathbf{m}_{v'}$ . The monitor  $M_3$  compares the received information about  $\mathbf{m}_2^j$  and  $\mathbf{m}_{v'}$  and decides whether  $\mathbf{m}_{v'}$  cancels the effect of  $\mathbf{m}_2^j$ . If  $\mathbf{m}_{v'}$  has not canceled the effect of  $\mathbf{m}_2^j$ ,  $M_3$  investigates the vector clocks of  $\mathbf{m}_2^j$  and  $\mathbf{m}_3^w$ . If  $\mathbf{m}_2^j$  has happened before  $\mathbf{m}_3^w$ , then  $M_3$  stores the result value “*Frm<sub>p</sub>*” or “*Frm*” sent by  $M_2$  to the history. If there is no relation between the vector clocks of  $\mathbf{m}_2^j$  and  $\mathbf{m}_3^w$ , it behaves conservatively and stores the “*Frm<sub>p</sub>*” result to its history. These scenarios will be continued to reach the message  $\mathbf{m}_{n-1}$  and the “*Frm*” or “*Frm<sub>p</sub>*” results values have been correctly propagated to the monitor of the message  $\mathbf{m}_{n-1}$  and hence declares the false verdict. ◀

► **Theorem 10.** *In a message-based system  $D = \{P_1 \dots P_n\}$ , no message sequence  $\omega \in \mathcal{L}(\mathcal{A})$  will be formed at runtime.*

**Proof.** We prove by contradiction: suppose that the unwanted message sequence  $\mathbf{m}_1 \dots \mathbf{m}_n$  is formed at runtime. Based on the second case of the proposed algorithm, the process  $P_n$  is blocked before sending the message  $\mathbf{m}_n$  and its corresponding monitor  $M_n$  inquires the sending status of  $\mathbf{m}_{n-1}$  from  $M_{n-1}$ . The process  $P_{n-1}$  cannot be blocked as  $\mathbf{m}_{n-1}$  is not the last message in the sequence. There are two cases depending on the response of  $M_{n-1}$ :

- (1) The message  $\mathbf{m}_{n-1}$  has not been sent: The monitor  $M_{n-1}$  responds to  $M_n$  that  $\mathbf{m}_{n-1}$  has not been sent and adds  $\mathbf{m}_{n-1}$  to *waitinglist* <sub>$n-1$</sub> . In this case, The message  $\mathbf{m}_{n-1}$  cannot be sent until  $\mathbf{m}_n$  has been sent and then  $M_n$  sends *notify* to  $M_{n-1}$ . As  $M_n$  finds that  $\mathbf{m}_{n-1}$  has not been sent, it informs  $P_n$  to send  $\mathbf{m}_n$  safely ((10) in Figure 6). After sending  $\mathbf{m}_n$ , the monitor  $M_n$  sends *notify* to  $M_{n-1}$  ((13) in Figure 6). Then,  $M_{n-1}$  removes  $\mathbf{m}_{n-1}$  from *waitinglist* <sub>$n-1$</sub>  and after that  $P_{n-1}$  can send  $\mathbf{m}_{n-1}$ . Hence, the message  $\mathbf{m}_n$  has been sent after  $\mathbf{m}_{n-1}$  and is contradicted by the assumption that  $\mathbf{m}_{n-1}\mathbf{m}_n$  is formed.
- (2) The message  $\mathbf{m}_{n-1}$  has been sent: The monitor  $M_{n-1}$  send the records of its history that are related to  $\mathbf{m}_{n-1}$  to  $M_n$ . By Lemma 1, the monitor  $M_{n-1}$  responds correctly if  $\mathbf{m}_1 \dots \mathbf{m}_{n-1}$  has been formed. There are two cases depending on the result of the received records: If there is any record which its result is *Frm* or *Frm<sub>p</sub>*,  $M_n$  finds that the sequence  $\mathbf{m}_1 \dots \mathbf{m}_{n-1}$  is formed. Hence, it informs  $P_n$  to send an error message instead of  $\mathbf{m}_n$ . So, the message  $\mathbf{m}_n$  has not been sent and the sequence  $\mathbf{m}_1 \dots \mathbf{m}_n$  is not formed and hence it is a contradiction. Otherwise, since there is no record with the result of *Frm* or *Frm<sub>p</sub>*,  $M_n$  finds that the sequence  $\mathbf{m}_1 \dots \mathbf{m}_{n-1}$  is not formed and informs  $P_n$  to send  $\mathbf{m}_n$  safely. So, the message  $\mathbf{m}_n$  has been sent as the sequence  $\mathbf{m}_1 \dots \mathbf{m}_{n-1}$  has not been formed. This is also contradicted by the formation of  $\mathbf{m}_1 \dots \mathbf{m}_{n-1}$ . ◀

# Broadcasting Competitively Against Adaptive Adversary in Multi-Channel Radio Networks

Haimin Chen

State Key Laboratory for Novel Software Technology, Nanjing University, China  
haimin.chen@smail.nju.edu.cn

Chaodong Zheng

State Key Laboratory for Novel Software Technology, Nanjing University, China  
chaodong@nju.edu.cn

---

## Abstract

Broadcasting in wireless networks is vulnerable to adversarial jamming. To thwart such behavior, *resource competitive analysis* is proposed. In this framework, sending, listening, or jamming on one channel for one time slot costs one unit of energy. The adversary can employ arbitrary strategy to disrupt communication, but has a limited energy budget  $T$ . The honest nodes, on the other hand, aim to accomplish broadcast while spending only  $o(T)$ . Previous work has shown, in a  $C$ -channels network containing  $n$  nodes, for large  $T$  values, each node can receive the message in  $\tilde{O}(T/C)$  time, while spending only  $\tilde{O}(\sqrt{T/n})$  energy. However, these multi-channel algorithms only work for certain values of  $n$  and  $C$ , and can only tolerate an oblivious adversary.

In this work, we provide new upper and lower bounds for broadcasting in multi-channel radio networks, from the perspective of resource competitiveness. Our algorithms work for arbitrary  $n, C$  values, require minimal prior knowledge, and can tolerate a powerful adaptive adversary. More specifically, in our algorithms, for large  $T$  values, each node's runtime is  $O(T/C)$ , and each node's energy cost is  $\tilde{O}(\sqrt{T/n})$ . We also complement algorithmic results with lower bounds, proving both the time complexity and the energy complexity of our algorithms are optimal or near-optimal (within a poly-log factor). Our technical contributions lie in using "epidemic broadcast" to achieve time efficiency and resource competitiveness, and employing coupling techniques in the analysis to handle the adaptivity of the adversary. At the lower bound side, we first derive a new energy complexity lower bound for 1-to-1 communication in the multi-channel setting, and then apply simulation and reduction arguments to obtain the desired result.

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

**Keywords and phrases** Broadcast, radio networks, resource competitive algorithms

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2020.22

**Related Version** Full paper is available on arXiv [9]: <https://arxiv.org/abs/2001.03936>.

**Funding** This work is supported by the Ministry of Science and Technology of China under Grant No.: 2018YFB1003200, and by the National Natural Science Foundation of China (NSFC) under Grant No.: 61702255 and 61672275.

**Acknowledgements** We would like to thank Weiming Feng and Guangxu Yang for the discussions.

## 1 Introduction

Consider a synchronous, time-slotted, single-hop wireless network formed by  $n$  devices (or, *nodes*). Each node is equipped with a radio transceiver, and these nodes communicate over a shared wireless medium containing  $C$  channels. In each time slot, each node can operate on one arbitrary channel, but cannot send and listen simultaneously. In this model, we study a fundamental communication problem – broadcasting – in which a designated *source* node wants to disseminate a message  $m$  to all other nodes in the network.



© Haimin Chen and Chaodong Zheng;

licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 22; pp. 22:1–22:16

Leibniz International Proceedings in Informatics



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

Lots of modern wireless devices are powered by battery and are able to switch between active and sleep states. Often, sending and listening occurring during active state dominate the energy expenditure, while sleeping costs much less [21]. Therefore, when running an algorithm, each node's energy complexity (or, *energy cost*) is often defined as the number of channel accesses [5–7, 17]; while time complexity is the number of slots till it halts.

The open and shared nature of wireless medium makes it vulnerable to jamming [16]. To thwart such behavior, one reasonable restriction is to bound the total amount of jamming, as injecting interfering signals also incurs operational cost. Specifically, we assume the existence of a jamming adversary called Eve. She can jam multiple channels in each slot, and jamming one channel for one slot costs one unit of energy. Eve has an energy budget  $T$  that is *unknown* to the nodes, and she can employ *arbitrary* strategy to disrupt communication.

This setting motivates the notation of *resource competitive algorithms* [1, 4, 8, 15, 18, 19] which focus on optimizing relative cost. Specifically, assume for each node the cost of sending or listening on one channel for one slot is one unit of energy (while idling is free),<sup>1</sup> can we design broadcast algorithms that ensure each node's cost is only  $o(T)$ ? Such results would imply Eve cannot efficiently stop nodes from accomplishing the distributed computing task in concern. Interestingly enough, the answer is positive. In particular, Gilbert et al. [15] present a resource competitive broadcast algorithm in the single-channel radio network setting: with high probability, each node receives the message and terminates within  $\tilde{O}(T + n)$  slots, while spending only  $\tilde{O}(\sqrt{T/n} + 1)$  energy.<sup>2</sup> This algorithm works even when Eve is adaptive and  $n$  is unknown to the nodes. Later, Chen and Zheng [8] consider the multi-channel setting: they show that when Eve is oblivious and  $C = O(n)$ , having multiple channels allows a linear speedup in time complexity, while the energy cost of each node remains to be  $\tilde{O}(\sqrt{T/n} + 1)$ .

In this paper, we develop two new multi-channel broadcast algorithms that can tolerate a stronger adaptive adversary and work for arbitrary  $n, C$  values, without sacrificing time efficiency or resource competitiveness. The first algorithm – called MULTICASTADP – needs to know  $n$ ; while the other more complicated one – called MULTICASTADVADP – does not. Both algorithms are randomized, and in the interesting case where  $T$  is large compared with  $n$  and  $C$ , each node's runtime is  $O(T/C)$ , while each node's energy cost is  $\tilde{O}(\sqrt{T/n})$ .<sup>3</sup>

► **Theorem 1.** *MULTICASTADP guarantees the following properties w.h.p.:* (a) all nodes receive the message and terminate within  $O(T/C + \tau_{time}) = \tilde{O}(T/C + \max\{n/C, C/n\})$  slots; and (b) the cost of each node is  $O(\sqrt{T/n} \cdot \sqrt{\lg T} \cdot \lg n + \tau_{cost}) = \tilde{O}(\sqrt{T/n} + C/n)$ .

■ When  $C = O(n)$ ,  $\tau_{time} = (n/C) \cdot \lg(n/C) \cdot \lg^2 n$ , and  $\tau_{cost} = \lg(n/C) \cdot \lg n$ .

■ When  $C = \Omega(n)$ ,  $\tau_{time} = (C/n) \cdot \lg(C/n) \cdot \lg^2 n$ , and  $\tau_{cost} = (C/n) \cdot \lg(C/n) \cdot \lg n$ .

► **Theorem 2.** *MULTICASTADVADP guarantees the following properties w.h.p.:* (a) all nodes receive the message and terminate within  $O(T/C + (nC + C^2) \cdot \lg^4(nC)) = \tilde{O}(T/C + nC + C^2)$  slots; and (b) the cost of each node is  $O(\sqrt{T/n} \cdot \lg^2 T + C^2 \cdot \lg^5(nCT) + (nC + C^2) \cdot \lg^4(nC)) = \tilde{O}(\sqrt{T/n} + nC + C^2)$ .

We also complement algorithmic results with lower bounds. Specifically, the  $O(T/C)$  term in runtime is optimal, as Eve can jam all  $C$  channels continuously for  $T/C$  slots. Meanwhile, the  $\tilde{O}(\sqrt{T/n})$  term in energy cost matches lower bound up to a poly-logarithmic factor. Thus our algorithms achieve (near) optimal time and energy complexity simultaneously.

<sup>1</sup> In reality, the cost for sending, listening, and jamming might differ, but they are often in the same order. The assumptions here are mostly for the ease of presentation, and are consistent with existing work. Moreover, allowing different actions to have different constant costs will not affect the results.

<sup>2</sup> We say an event happens *with high probability (w.h.p.)* if the event occurs with probability at least  $1 - 1/n^c$ , for some tunable constant  $c \geq 1$ . Moreover, we use  $\tilde{O}$  to hide poly-log factors in  $n, C$ , and  $T$ .

<sup>3</sup> The primary goal of resource competitive algorithms is to optimize nodes' cost for large  $T$  values, see previous work (e.g., [4, 18]) and discussion on resource competitiveness in Section 1.2 for more details.

► **Theorem 3.** *For an adaptive adversary with budget  $T$ , any fair multi-channel broadcast algorithm that succeeds with constant probability imposes an expected cost of  $\Omega(\sqrt{T/n})$  per node. Notice, an algorithm is fair if all participating nodes have the same expected cost; both *MULTICASTADP* and *MULTICASTADVADP* are fair.*

## 1.1 Related Work

Broadcasting in radio networks is non-trivial due to collisions. Classical results often rely on variants of the DECAY procedure [3], while recent ones (e.g., [10, 14]) tend to employ more advanced techniques (e.g., network decomposition) to improve performance. Besides time complexity, energy cost has also been taken into consideration when building communication primitives (e.g., [5–7, 13]), but usually without assuming the existence of a jamming adversary.

Distributed computing in jamming-prone environment has attracted a lot of attention as well. Researchers from the theory community usually pose certain restrictions on the behavior of the malicious user(s), and then develop corresponding countermeasures (e.g., [2, 11, 20, 22]). Unfortunately, these restrictions somewhat limit the adversary’s strategy, and many of the proposed algorithms also require honest nodes to spend a lot of energy. In view of these, *resource competitive analysis* [4] is proposed. This framework allows more flexibility for the adversary, hence potentially better captures reality. However, it also brings new challenges to the design and analysis of algorithms.

In 2011, King, Saia, and Young [19] developed the first resource competitive algorithm, in the context of 1-to-1 communication. (That is, Alice wants to send a message to Bob.) Specifically, the proposed Las Vegas algorithm ensures the expected cost of Alice and Bob is only  $O(T^{0.62} + 1)$ . As mentioned earlier, Gilbert et al. [15] later devise a single-channel broadcast algorithm that is resource competitive against jamming. They have also proved several lower bounds showing the algorithm’s energy cost is near optimal. The work that is most closely related to ours is by Chen and Zheng [8], in which several multi-channel broadcast algorithms are developed. However, an important drawback of [8] is that it only considers an oblivious adversary, while all other previous results can tolerate an adaptive (or even reactive) adversary. In this paper, we close the gap by considering an adaptive adversary, and provide similar or better results than [8] that work for arbitrary values of  $n$  and  $C$ . We also prove our results are (near) optimal by deriving new lower bounds.

## 1.2 Additional Model Details

All nodes in the network start execution simultaneously and can independently generate random bits. In each slot, each node either sends a message on a channel, or listens on a channel, or remains idle. Only listening nodes get feedback regarding channel status. The adversary Eve is adaptive: at the beginning of each slot, she is given all past execution history and can use these information to determine her behavior. However, she does *not* know honest nodes’ random bits or behavior of the current slot.

In each slot, for each listening node, the channel feedback is determined by the number of sending nodes on that channel and the behavior of Eve. Specifically, consider a slot and a channel  $ch$ . If no node sends on  $ch$  and Eve does not jam  $ch$ , then nodes listening on  $ch$  hear silence. If exactly one node sends a message on  $ch$  and Eve does not jam  $ch$ , then nodes listening on  $ch$  receive the unique message. Finally, if at least two nodes send on  $ch$  or Eve jams  $ch$ , then nodes listening on  $ch$  hear noise. Note that we assume nodes cannot tell whether noise is due to jamming or message collision (or both).

We adopt the following definition of resource competitive algorithms introduced in [4]:

► **Definition 4.** Consider an execution  $\pi$  in which nodes execute algorithm  $\mathcal{A}_N$  and Eve employs strategy  $\mathcal{A}_E$ . Let  $\text{cost}_u(\pi)$  denote the energy cost of node  $u$ , and  $T(\pi)$  denote the energy cost of Eve. We say  $\mathcal{A}_N$  is  $(\rho, \tau)$ -resource competitive if  $\max_u \{\text{cost}_u(\pi)\} \leq \rho(T(\pi)) + \tau$  for any execution  $\pi$ .

In above,  $\rho$  is a function of  $T$  and possibly other parameters (such as  $n, C$ ). It captures the additional cost nodes incur due to jamming. The other function  $\tau$  captures the cost of the algorithm when Eve is absent, thus  $\tau$  should not depend on  $T$ . Most resource competitive algorithms aim to minimize  $\rho$ , while keeping  $\tau$  reasonably small.

### 1.3 Overview of Techniques

**Fast and competitive broadcast against jamming.** Most resource competitive broadcast algorithms group slots into consecutive *epochs*, and execute a jamming-resistant broadcast scheme within each epoch. In the single-channel setting, often the core idea is to broadcast “sparsely” [18, 19]. Consider 1-to-1 communication as an example. If both nodes send and listen in  $\Theta(\sqrt{R})$  random slots in an epoch of length  $R$ , then by a birthday-paradox argument, successful transmission will occur with constant probability even if Eve jams constant fraction of all  $R$  slots. In the multi-channel setting, “*epidemic broadcast*” is employed [8]. In the simplest form of this scheme, in each time slot, each node will choose a random channel from  $[C] = \{1, 2, \dots, C\}$ . Then, each informed node (i.e., the node knows the message  $m$ ) will broadcast  $m$  with a constant probability, while each uninformed node will listen with a constant probability. If  $C = n/2$ , broadcast will complete in  $O(\lg n)$  slots w.h.p., and this claim holds even if Eve jams constant fraction of all channels for constant fraction of all slots.

In designing MULTICASTADP and MULTICASTADVADP, one key challenge is to extend the basic epidemic broadcast scheme to guarantee an optimal  $O(T/C)$  runtime for arbitrary  $n, C$  values, without increasing energy expenditure. To that end, we note that in the single-channel setting, [15] has shown  $\Theta(1/\sqrt{Rn})$  is roughly an optimal working probability (i.e., sending/listening probabilities). When  $C$  channels are available, a good way to adjust the probability would be to multiply it by a factor of  $\sqrt{C}$  (i.e.,  $\Theta(\sqrt{C}/(Rn))$ ). Intuitively, the reason being: if each node works on  $\sqrt{C}$  random channels simultaneously in each slot, then again by a birthday-paradox argument, each pair of nodes will meet on at least one channel with at least constant probability, which effectively means the optimal single-channel analysis could be applied again. Of course nodes do not have multiple transceivers and cannot work on multiple channels simultaneously, but over a period of time, multiplying the single-channel working probability by  $\sqrt{C}$  achieves similar effect. On the other hand, although the working probability of nodes is increased by a factor of  $\sqrt{C}$ , the energy expenditure of Eve will increase by a factor of  $\Theta(C)$ . As a result, compared with single-channel solutions, our algorithms have a  $\Theta(C)$  speedup in time, yet the resource competitive ratio is unchanged.

**Termination and the coupling technique.** Termination mechanism is another key integrant, it ensures nodes stop execution correctly and timely. For each node  $u$ , a helpful termination criterion is comparing  $N_u$  – the number of silent slots it observed during the current epoch – to some pre-defined threshold. To argue the correctness of our algorithms, we often need to show  $N_u$  is close to its expected value. However, this is non-trivial if Eve is adaptive.

To see this, consider an epoch containing  $R$  slots. Define  $G_i$  as the *behavior* (i.e., channels choices and actions) of all nodes in slot  $i$ , and define  $Q_i$  – the set of channels that are *not* jammed by Eve – as the *jamming result* of slot  $i$ . Note that  $N_u$  can be written as the sum of  $R$  indicator random variables:  $N_u = \sum_{i=1}^R N_{u,i}$ , where  $N_{u,i} = 1$  iff  $u$  hears silence in the  $i^{\text{th}}$  slot.  $N_{u,i}$  is determined by  $G_i$  and  $Q_i$ , but in general  $Q_i$  can be arbitrary function of  $\{G_1, G_2, \dots, G_{i-1}, Q_1, Q_2, \dots, Q_{i-1}\}$ . Nonetheless, in case Eve is oblivious (i.e., an offline



adversary), her optimal strategy would be a fixed vector of jamming results  $\langle q_1, q_2, \dots, q_R \rangle$ , thus  $\{N_{u,1}, N_{u,2}, \dots, N_{u,R}\}$  are mutually independent when  $\{G_1, G_2, \dots, G_R\}$  are mutually independent (this can be easily enforced by the algorithm). Therefore, if Eve is oblivious, we can directly apply powerful concentration inequalities like Chernoff bounds to show  $N_u$  is close to its expectation. However, once Eve becomes adaptive,  $Q_i$  could depend on  $\{G_1, \dots, G_{i-1}\}$  and above observations no longer hold:  $\{N_{u,1}, \dots, N_{u,R}\}$  could be dependent!

In this paper, we leverage the *coupling* technique (see, e.g., [12]) extensively to resolve the dependency issue. Specifically, for each vector of jamming results over one epoch, we create a coupled execution and relate  $N_u$  to a corresponding random variable in the coupled execution. By carefully crafting the coupling, the random variable in the coupled execution can be interpreted as the sum of a set of independent random variables, allowing us to bound the probability that  $N_u$  deviates a lot from its expectation. However, there is a catch in this approach: bounding the probability that  $N_u$  deviates a lot from its expectation requires us to sum the failure probability over all jamming results vectors, but there may be  $\Theta(2^{CR})$  such vectors! Our solution to this new problem is to group all vectors into fewer categories, so that vectors within one category have identical or similar effects on the metric we concern.

► **Remark.** Techniques like “principle of deferred decision”, or the ones used in previous work, cannot resolve the dependency issue directly in our setting. See full version of our paper for more discussion.

**Lower bound.** Existing result [15] indicates fair broadcast in the single-channel settings requires each node spending  $\Omega(\sqrt{T/n})$  energy, but could it be the case that having multiple channels also reduces the energy complexity of the problem? We show the answer is negative.

Specifically, for any multi-channel broadcast algorithm  $\mathcal{A}_n$ , we devise a corresponding multi-channel 1-to-1 communication algorithm  $\mathcal{A}_2$  that simulates  $\mathcal{A}_n$  internally. We also devise a jamming strategy  $\mathcal{S}$  for disrupting  $\mathcal{A}_n$  and  $\mathcal{A}_2$ : in each slot, for each channel, Eve jams that channel iff a successful transmission will occur on that channel with a probability exceeding  $1/T$ .  $\mathcal{A}_2$  and  $\mathcal{S}$  are carefully constructed so that algorithms’ success probabilities and nodes’ energy expenditure in the two executions (i.e., in  $(\mathcal{A}_n, \mathcal{S})$  and  $(\mathcal{A}_2, \mathcal{S})$ ) are closely connected. Then, we derive an energy complexity lower bound for multi-channel 1-to-1 communication assuming Eve uses  $\mathcal{S}$ . (This result, Theorem 17 in Section 7, could be of independent interest and is strong in two aspects: (a) the bound holds even if the two nodes has multiple transceivers; (b) its proof uses a novel approach to handle adaptive Monte Carlo algorithms.) Finally, an energy complexity lower bound for  $\mathcal{A}_n$  is obtained via reduction.

## 2 Notations

Let  $V$  be the set of all nodes. Since all algorithms developed in this paper proceed in epochs, consider a slot  $i$  in an epoch of length  $R$ , where  $1 \leq i \leq R$ . Denote  $Q_i \in 2^{[C]}$  as the jamming result of the  $i^{\text{th}}$  slot:  $Q_i$  is the set of channels that are *not* jammed by Eve in the  $i^{\text{th}}$  slot. Denote  $G_i = \langle (G_{i,v}^{ch})_{v \in V}, (G_{i,v}^{act})_{v \in V} \rangle$  as the behavior (i.e., channel choices and actions) of the  $n$  nodes in the  $i^{\text{th}}$  slot:  $G_i \in \Omega = [C]^n \times \{\text{send, listen, idle}\}^n$ .<sup>4</sup> Since Eve is adaptive,  $Q_i$  may depend on  $\mathbf{G}_{<i} = (G_1, \dots, G_{i-1})$ . Lastly, define  $\mathbf{Q}_{\leq i} = (Q_1, \dots, Q_i)$ .

<sup>4</sup> There is a technical subtlety worth clarifying. The “behavior” here does not care about the exact content to be broadcast if some node(s) choose to send message(s) in a slot. That is, for each slot, the “behavior” here is *not* some element in  $[C]^n \times (M \cup \{\text{listen, idle}\})^n$ , where  $M$  is the set of all possible messages. This is for the ease of presentation and will not affect the correctness of our results.



To quantify the severity of jamming from Eve, for a given slot, we use  $\mathcal{E}(> x)$  (respectively,  $\mathcal{E}(\geq x)$ ,  $\mathcal{E}(< x)$ ,  $\mathcal{E}(\leq x)$ ) to denote that in a slot, more than (respectively, at least, less than, at most)  $x$  fraction of the  $C$  channels are *not* jammed by Eve. In the following, we use  $\mathcal{E}(\cdot x)$  to represent one of the above four forms. (I.e., “ $\cdot$ ” denotes “ $>$ ”, “ $\geq$ ”, “ $<$ ”, or “ $\leq$ ”.)

For an epoch, we use  $\mathcal{E}(>^y)(\cdot x)$  (respectively,  $\mathcal{E}(\geq^y)(\cdot x)$ ,  $\mathcal{E}(<^y)(\cdot x)$ ,  $\mathcal{E}(\leq^y)(\cdot x)$ ) to denote the event that for more than (respectively, at least, less than, at most)  $y$  fraction of the  $R$  slots,  $\mathcal{E}(\cdot x)$  happen. For example,  $\mathcal{E}(>^{0.1})(> 0.2)$  means in an epoch, for more than 0.1 fraction of all slots, Eve leaves more than 0.2 fraction of all channels unjammed.

Define negation operation in the following manner:  $\overline{(> x)} = (\leq x)$  and vice versa;  $\overline{(< x)} = (\geq x)$  and vice versa. Further define complement operation in the following manner:  $\mathcal{C}(> x) = (< 1 - x)$  and vice versa;  $\mathcal{C}(\geq x) = (\leq 1 - x)$  and vice versa. It is easy to verify  $\mathcal{E}(>^y)(\cdot x) = \mathcal{E}(\mathcal{C}(>^y))(\overline{\cdot x})$  and  $\mathcal{E}(\cdot x) = \mathcal{E}(\overline{\cdot x})$ . Therefore:

$$\overline{\mathcal{E}(\geq^y)(\geq x)} = \mathcal{E}(\leq^{1-y})(\geq \overline{x}) = \mathcal{E}(>^{1-y})(\geq \overline{x}) = \mathcal{E}(>^{1-y})(< x)$$

Again, as a simple example, the above equality implies “if in an epoch, it is not the case that in at least 0.1 fraction of all slots Eve leaves at least 0.2 fraction of all channels unjammed, then it must be the case that in more than 0.9 fraction of all slots, Eve leaves less than 0.2 fraction of all channels unjammed; and vice versa”. (I.e.,  $\overline{\mathcal{E}(\geq^{0.1})(\geq 0.2)} = \mathcal{E}(>^{0.9})(< 0.2)$ .)

### 3 The MultiCastAdp Algorithm

Each node  $u$  maintains a Boolean variable  $M_u$  to indicate whether it knows the message  $m$  (in which case  $M_u$  is *true* and  $u$  is *informed*) or not (in which case  $M_u$  is *false* and  $u$  is *uninformed*). Initially, only the source node sets  $M_u = \text{true}$ . The algorithm proceeds in epochs and the  $i^{\text{th}}$  epoch contains  $R_i = a \cdot 4^i \cdot i \cdot \lg^2 n$  slots, where  $a$  is some large constant. In each slot in epoch  $i$ , for each node  $u$  that is still executing the algorithm (i.e., the node is still *active*), it will hop to a uniformly chosen random channel. Then,  $u$  will choose to broadcast or listen each with probability  $p_i = (\sqrt{C/n})/2^i$ . If  $u$  decides to broadcast and  $M_u = \text{true}$ , it sends  $m$ ; otherwise,  $u$  sends a special beacon message  $\pm$ . On the other hand, if in a slot  $u$  decides to listen, it will record the channel feedback. Finally, by the end of an epoch  $i$ , for a node  $u$ , if among the slots it listened within this epoch, at least  $(p_i R_i)/2$  are silent slots, then  $u$  will halt. One point worth noting is, the first epoch number is not necessarily one; instead, it is chosen as a sufficiently large integer to ensure  $p_i \leq 1/2$  and  $p_i \leq C/(4n)$ . Hence, the first epoch number is  $I_b = 2 + \lceil \max\{\lg(\sqrt{n/C}), \lg(\sqrt{C/n})\} \rceil$ . Complete pseudocode of MULTICASTADP is provided in the full version of the paper.

### 4 Analysis of MultiCastAdp

**Effectiveness of epidemic broadcast.** The first technical lemma states if in an epoch jamming from Eve is not strong and every node is active, then all nodes will be informed by the end of the epoch. More specifically:

► **Lemma 5.** *If all nodes are active at the beginning of epoch  $i$ , and during epoch  $i$  event  $\mathcal{E}^{\geq y_1}(\geq x_1)$  occurs, then by the end of this epoch, all nodes will be informed, with probability at least  $1 - n^{-\Theta(i)}$ . Here,  $x_1 = y_1 = 0.1$ , and  $\mathcal{E}^{\geq y_1}(\geq x_1)$  is defined in Section 2.*

This lemma highlights the effectiveness of the epidemic broadcast scheme. Intuitively, it holds because when less than  $n/2$  nodes know message  $m$ , the number of informed nodes will increase by some constant factor every so often; and once at least  $n/2$  nodes know  $m$ ,

remaining uninformed nodes will quickly learn the message too. To prove this intuition rigorously, however, we need to apply the coupling technique.

To construct the coupling, we first specify how nodes' behavior is generated. Fix an epoch, imagine two sufficiently long bit strings  $\mathbf{T}_{high}$  and  $\mathbf{T}_{low}$  in which each bit is generated independently and uniformly at random. Divide  $\mathbf{T}_{high}$  and  $\mathbf{T}_{low}$  into consecutive *chunks* of equal size, such that each chunk provides enough random bits for  $n$  nodes to determine their behavior in a slot. More formally,  $\mathbf{T}_{high} = (T_{hi}^{(1)}, T_{hi}^{(2)}, \dots, T_{hi}^{(R)})$  and  $\mathbf{T}_{low} = (T_{lo}^{(1)}, T_{lo}^{(2)}, \dots, T_{lo}^{(R)})$ , where each  $T_{hi}^{(*)}$  or  $T_{lo}^{(*)}$  is a chunk. Next, we introduce three processes that are used during the coupling:  $\beta$ ,  $\beta'$ , and  $\gamma$ .

We begin with  $\beta$ , which is an execution of MULTICASTADP with adversary Eve. The tricky part about  $\beta$  is: in the  $i^{\text{th}}$  slot, nodes' behavior  $G_i$  is *not* determined by  $T_{hi}^{(i)}$  or  $T_{lo}^{(i)}$  directly. Instead, it is generated in a more complicated way. Specifically, at the beginning of slot  $i$ , Eve first computes its jamming result  $Q_i$  (i.e., the set of unjammed channels) based on  $\mathbf{Q}_{<i}$  and  $\mathbf{G}_{<i}$ . If  $|Q_i| \geq x_1 C$  and the number of previously used chunks from  $\mathbf{T}_{high}$  is no more than  $y_1 R$ , then we pick the next unused chunk from  $\mathbf{T}_{high}$ ; otherwise, we pick the next unused chunk from  $\mathbf{T}_{low}$ . Assume  $T^{(j)}$  is the chosen chunk, and it computes to nodes' behavior  $\langle (\hat{G}_v^{ch})_{v \in V}, (\hat{G}_v^{act})_{v \in V} \rangle$ . Still, we do not use  $\langle (\hat{G}_v^{ch})_{v \in V}, (\hat{G}_v^{act})_{v \in V} \rangle$  as nodes' behavior. Instead, we permute the channel choices according to the jamming result. Specifically, for each  $q \in 2^{[C]}$ , define permutation  $\pi_q$  on  $[C]$  as follows: for  $1 \leq k \leq |q|$ ,  $\pi_q(k)$  is the  $k^{\text{th}}$  smallest element in  $q$ ; and for  $|q| + 1 \leq k \leq C$ ,  $\pi_q(k)$  is the  $(k - |q|)^{\text{th}}$  smallest element in  $[C] \setminus q$ . (For example, if  $C = 5$  and  $q = \{2, 4\}$ , then  $\pi_q$  permutes  $\langle 1, 2, 3, 4, 5 \rangle$  to  $\langle 2, 4, 1, 3, 5 \rangle$ .) Further define bijection  $\Psi_q : \Omega \rightarrow \Omega$  using  $\pi_q$ :

$$\Psi_q \left( \left\langle \left( \hat{G}_v^{ch} \right)_{v \in V}, \left( \hat{G}_v^{act} \right)_{v \in V} \right\rangle \right) = \left\langle \left( \pi_q \left( \hat{G}_v^{ch} \right) \right)_{v \in V}, \left( \hat{G}_v^{act} \right)_{v \in V} \right\rangle$$

Now, we use  $\langle (\pi_q(\hat{G}_v^{ch}))_{v \in V}, (\hat{G}_v^{act})_{v \in V} \rangle$  as nodes' behavior  $G_i$  in slot  $i$ . Formally, let  $K(\mathbf{Q}_{\leq i}) = \sum_{j=1}^i \mathbb{I}[|Q_j| \geq x_1 C]$  count the number of weakly jammed slots (i.e.,  $|Q_j| \geq x_1 C$ ) among the first  $i$  slots, where each  $\mathbb{I}[|Q_j| \geq x_1 C]$  is an indicator random variable. Then,  $G_i$  can be defined as:

$$G_i = \begin{cases} \Psi_{Q_i} \left( T_{hi}^{(K(\mathbf{Q}_{\leq i}))} \right), & |Q_i| \geq x_1 C \text{ and } K(\mathbf{Q}_{\leq i}) \leq y_1 R \\ \Psi_{Q_i} \left( T_{lo}^{(i-K(\mathbf{Q}_{\leq i}))} \right), & |Q_i| < x_1 C \text{ and } K(\mathbf{Q}_{\leq i}) \leq y_1 R \\ \Psi_{Q_i} \left( T_{lo}^{(i-y_1 R)} \right), & \text{otherwise} \end{cases}$$

Careful readers might suspect does  $\mathbf{G} = (G_1, G_2, \dots, G_R)$  in process  $\beta$  really has the correct distribution  $\mathcal{G}$  we want. (That is,  $\mathcal{G}$  is the distribution in which the behavior of the nodes are determined by, say  $\mathbf{T}_{low}$ , directly.) After all, just by looking at the definition, it seems  $G_i$  depends on  $Q_i$ , which is controlled by Eve. Interestingly enough, indeed  $\mathbf{G} \sim \mathcal{G}$ . To understand this intuitively, consider the following simple game played between Alice and Eve. In each round, Alice tosses a fair coin but does not reveal it to Eve (this coin plays similar role as  $T^{(j)}$ ). However, Eve can decide whether to flip the coin or not (this is like permuting channel assignments according to  $q$ ). Finally, the coin is revealed and the game continues into the next round. Now, a simple but important observation is: the coin is still a fair coin in each round, although Eve can decide whether to flip it or not. Similarly, back to our setting, we can show  $\mathbf{G} \sim \mathcal{G}$ .

We continue to introduce process  $\beta'$ . In  $\beta'$ , still there are  $n$  nodes executing MULTICASTADP, along with a jamming adversary Carlo. However, for each slot  $i$ , if in  $\beta$  nodes use  $\Psi_{Q_i}(T_{hi}^{(j)})$  (resp.,  $\Psi_{Q_i}(T_{lo}^{(j)})$ ) to determine their behavior, then in  $\beta'$  nodes directly use  $T_{hi}^{(j)}$  (resp.,  $T_{lo}^{(j)}$ ), and Carlo leaves channels  $\{1, 2, \dots, x_1 C\}$  unjammed (resp., jams all channels).

Finally, in  $\gamma$ , again there are  $n$  nodes executing MULTICASTADP, yet the adversary uses a fixed strategy: in the first  $y_1R$  slots, channels  $\{1, 2, \dots, x_1C\}$  are unjammed; and in the remaining  $(1 - y_1)R$  slots, all channels are jammed. Besides, in the  $i^{\text{th}}$  slot, nodes directly use chunk  $T_{hi}^{(i)}$  to compute their behavior if  $i \leq y_1R$ , and use chunk  $T_{lo}^{(i-y_1R)}$  otherwise.

We are now ready to sketch the proof of Lemma 5. (Complete proofs can be found in the full version of the paper.)

**Proof sketch of Lemma 5.** Define  $\mathcal{E}_X$  (respectively,  $\mathcal{E}_{X'}$ , and  $\mathcal{E}_Y$ ) be the event that some node is still uninformed by the end of process  $\beta$  (respectively,  $\beta'$ , and  $\gamma$ ). Let  $\mathcal{E}_{\geq}$  be event  $\mathcal{E}^{\geq y_1}(\geq x_1)$ . The following claims capture the relationship between these events:

*Claim I:  $\mathcal{E}_X$  implies  $\mathcal{E}_{X'}$ .* Consider a slot  $i$  and two nodes  $u$  and  $v$ , assume in  $\beta$  node  $u$  broadcasts on channel  $ch$  and node  $v$  listens on  $ch$ . Then, by definition of our permutation function  $\pi$ , in that same slot in  $\beta'$ ,  $u$  must broadcast on  $\pi_{Q_i}^{-1}(ch)$  and  $v$  must listen on  $\pi_{Q_i}^{-1}(ch)$ . We argue a failed transmission attempt on  $ch$  in  $\beta$  will also fail on  $\pi_{Q_i}^{-1}(ch)$  in  $\beta'$ : (a) if some third node  $w$  also broadcasts on  $ch$  in slot  $i$  in  $\beta$ , then  $w$  must also broadcast on  $\pi_{Q_i}^{-1}(ch)$  in slot  $i$  in  $\beta'$ ; (b) if Eve jams  $ch$  in slot  $i$  in  $\beta$ , then Carlo must also jam  $\pi_{Q_i}^{-1}(ch)$  in slot  $i$  in  $\beta'$ . Thus, assuming  $v$  is uninformed in both  $\beta$  and  $\beta'$  at the beginning of slot  $i$ , then by the end of slot  $i$ , if  $v$  is still uninformed in  $\beta$ , it must be the case that  $v$  is also uninformed in  $\beta'$ . A simple induction immediately leads to the claim.

*Claim II:  $(\mathcal{E}_{X'} \wedge \mathcal{E}_{\geq})$  implies  $\mathcal{E}_Y$ .* If  $\mathcal{E}_{\geq}$  happens in  $\beta$ , then in both  $\beta'$  and  $\gamma$ , Eve leaves channels  $\{1, 2, \dots, x_1C\}$  unjammed in  $y_1R$  slots, and jams all channels in remaining slots. Observe that we can ignore the slots in which all channels are jammed; and in each remaining slot, nodes' behavior and channel feedback are identical in the two processes.

Therefore,  $\Pr[\mathcal{E}_X \wedge \mathcal{E}_{\geq}] \leq \Pr[\mathcal{E}_Y]$ . The effectiveness of the epidemic broadcast scheme is easy to demonstrate in process  $\gamma$ , as the jamming strategy of the adversary in  $\gamma$  is not adaptive. Specifically, we conclude  $\Pr[\mathcal{E}_Y] \leq \exp(-\Theta(i \cdot \lg n))$ . ◀

**Competitiveness and Correctness.** We prove two other key lemmas in this part. The first one shows Eve cannot stop nodes from halting without spending a lot of energy, thus guaranteeing the resource competitiveness of the termination mechanism.

► **Lemma 6.** *Fix an epoch  $i$  and a node  $u$ , assume  $u$  is alive at the beginning of this epoch. By the end of this epoch, with probability at most  $\exp(-\Theta(i \cdot \lg^2 n))$ , the following two events happen simultaneously: (a)  $\mathcal{E}^{\geq y_2}(\geq x_2)$  occurs during the epoch; and (b) node  $u$  does not halt. Here,  $x_2 = y_2 = 0.99$ , and  $\mathcal{E}^{\geq y_2}(\geq x_2)$  is defined in Section 2.*

**Proof sketch.** Arrange the randomness of nodes as what we do in the proof of Lemma 5, except that we use parameter  $x_2 = 0.99$  to replace  $x_1$ , and  $y_2 = 0.99$  to replace  $y_1$ . Let  $R$  be the length of the epoch,  $p$  be nodes' working probability, and  $\mathcal{E}_{\geq}$  be event  $\mathcal{E}^{\geq y_2}(\geq x_2)$ . Define  $X_i$  (respectively,  $X'_i$ , and  $Y_i$ ) be an indicator random variable taking value one iff  $u$  hears silence in the  $i^{\text{th}}$  slot in  $\beta$  (respectively,  $\beta'$ , and  $\gamma$ ). Following random variables are what we intend to couple:  $X = \sum_{i=1}^R X_i$ ,  $X' = \sum_{i=1}^R X'_i$ , and  $Y = \sum_{i=1}^R Y_i$ . Specifically:

*Claim I: For any integer  $t \geq 0$ ,  $\Pr[X \leq t] \leq \Pr[X' \leq t]$ .* Similar to the proof of Claim I in the proof of Lemma 5, for each slot, if in that slot  $u$  hears silence in  $\beta'$ , then by definition of our permutation function  $\pi$  and the construction of  $\beta$  and  $\beta'$ , it must be the case that  $u$  also hears silence in  $\beta$ . Thus,  $X'_i = 1$  implies  $X_i = 1$ , resulting in  $X' \leq X$ .

*Claim II: For any integer  $t \geq 0$ ,  $\Pr[(X' \leq t) \wedge \mathcal{E}_{\geq}] \leq \Pr[Y \leq t]$ .* If  $\mathcal{E}_{\geq}$  happens in  $\beta'$ , then in both  $\beta'$  and  $\gamma$ , Eve leaves channels  $\{1, 2, \dots, x_2C\}$  unjammed for  $y_2R$  slots, and jam all channels in other slots. Note that in each of the  $R$  slots, nodes' behavior are independent and are sampled from an identical distribution, so the indices of the  $y_2R$  slots does not matter.

Therefore,  $\Pr[(X \leq t) \wedge \mathcal{E}_{\geq}] \leq \Pr[Y \leq t]$ . Since  $\{Y_1, Y_2, \dots, Y_R\}$  is a set of mutually independent random variables, bounding  $\Pr[Y < Rp/2]$  is easy. Specifically,  $\mathbb{E}[Y] = y_2 \cdot x_2 \cdot p \cdot (1 - p/C)^{n-1} \geq 0.99^2 \cdot Rp \cdot (1 - p/C)^n \geq 0.99^2 \cdot Rp \cdot e^{-2np/C} \geq 0.99^2 \cdot Rp \cdot e^{-0.5} > 0.59Rp$ . Apply a Chernoff bound, we know  $\Pr[Y < Rp/2] \leq \exp(-\Theta(Rp)) \leq \exp(-\Theta(i \cdot \lg n))$ . ◀

The second lemma states that all nodes must have been informed before any node decides to halt, thus message dissemination must have completed before any node stops execution. To prove the lemma, we consider two complement cases: either Eve jams a lot in the epoch, or she does not. If jamming is not strong, Lemma 5 implies no node remains uninformed. Otherwise,  $u$  should not hear a lot of silent slots and will not halt. Notice, handling the strong jamming case also relies on the coupling technique.

► **Lemma 7.** *Fix an epoch  $i$  in which all nodes are active, fix a node  $u$ . By the end of this epoch, with probability at most  $\exp(-\Theta(i \cdot \lg n))$ , the following two events happen simultaneously: (a) node  $u$  halts; and (b) some node is still uninformed.*

**Main theorem.** We sketch the proof of Theorem 1 in this last part.

Fix a node  $u$ , we begin by computing how long  $u$  remains active. Let  $L$  be the total runtime of  $u$ . Since epoch length increases geometrically, we only need to focus on the last epoch in which  $u$  is active. Also, notice that Lemma 6 suggests Eve must jam a lot in an epoch – the amount of which can be described as some function of epoch length – to stop  $u$  from halting. Putting these pieces together, we show  $\Pr(L > \Theta(1) \cdot T/C) \leq n^{-\Omega(1)}$ . By a union bound, we know when  $T = \Omega(C)$  w.h.p. all nodes halt within  $O(T/C)$  slots.

Next, we analyze the cost of nodes. Again fix a node  $u$ , let  $F$  denote its total cost. By an argument similar to above, we are able to prove  $\Pr(F > \Theta(\lg n) \cdot \sqrt{\lg T \cdot (T/n)}) \leq n^{-\Omega(1)}$ . By a union bound, we know when  $T = \Omega(C)$  w.h.p. the cost of each node is  $O(\sqrt{T/n} \cdot \sqrt{\lg T} \cdot \lg n)$ .

The last step is to show with high probability each node must have been informed when it halts, and this can be proved via an application of Lemma 7.

Finally, we note that when  $T = o(C)$ , all nodes will halt by the end of the first epoch, with high probability. This results in the  $\tau_{time}$  and  $\tau_{cost}$  terms in the theorem statement.

## 5 The MultiCastAdvAdp Algorithm

Our second algorithm – called MULTICASTADVADP – works even if knowledge of  $n$  is absent. However, its design and analysis are much more involved than that of MULTICASTADP.

**Building MultiCastAdvAdp.** When the value of  $n$  is unknown, the principal obstacle lies in properly setting nodes’ working probabilities. In view of this, we let MULTICASTADVADP contain multiple *super-epochs*, each of which contains multiple *phases*, and nodes may use different working probabilities in different phases. Notice, for each super-epoch, we need to ensure it contains sufficiently many “good” phases, in the sense that within each such good phase broadcast will succeed if Eve does not heavily jam it. Another challenge posed by the unknown  $n$  value is that the simple termination criterion – large fraction of silent slots – no longer works, as this can happen when the working probability is too low.

Gilbert et al. [15] provide a solution to the above two challenges in the single-channel setting. Specifically, at the beginning of a super-epoch  $i$ , nodes set their initial working probability to a pre-defined small value. After each phase, each node  $u$  increases its working probability  $p_u$  by a factor of  $2^{\max\{0, \eta_u - 0.5\}/i}$ , where  $\eta_u$  denotes the fraction of silent slots  $u$  observed within the phase. This mechanism provides two important advantages: (a) Eve has

to keep jamming heavily to prevent  $p_u$  from reaching the ideal value; and (b)  $p_u$  and  $p_v$  might be different for two nodes  $u$  and  $v$ , but the difference is bounded. As for termination, the number of messages nodes heard could be a good metric. However, a simple threshold would not work. Instead, Gilbert et al. develop a two-stage termination mechanism: when a node  $u$  hears the message sufficiently many times, it becomes a **helper** and obtains an estimate of  $n$ ; Later, when  $u$  is sure that all nodes have become **helper**, it will stop execution.

In MULTICASTADVADP, we extend the above approach to the multi-channel setting. Specifically, we observe that the single-channel message dissemination scheme used in [15] is relatively slow in that it needs  $\Theta(\lg n)$  phases to accomplish broadcast. By contrast, in MULTICASTADVADP, the application of epidemic broadcast reduces this time period to a single weakly-jammed phase. This replacement is not a simple cut-and-paste. Instead, we also adjust the phase structure accordingly. In particular, each phase now contains two *steps*. This adjustment further demands us to change the way nodes' update their working probabilities after each phase:  $p_u \leftarrow p_u \cdot 2^{\max\{0, \eta_u^{\text{step1}} + \eta_u^{\text{step2}} - 1.5\}}$ . In the end, MULTICASTADVADP provides a slightly better resource competitive ratio than [15].

Handling adaptivity via coupling also becomes more challenging. In more detail, in each phase we need the number of silent slots  $u$  heard  $N_u$  to be close to its expectation for *any* jamming results vector (instead of, say, only when jamming is strong, as in the proof of Lemma 7). To acquire the desired results, we have to consider jamming results vectors at a much finer level (rather than a single category, as in the proof of Lemma 6 and Lemma 7), which in turn requires the failure probability for each category to be much lower (otherwise a union bound over the increased number of categories would not work). Allowing  $N_u$  to have larger deviation from its expectation solves the issue, but it further demands the initial working probability nodes used at the beginning of each epoch to be sufficiently high. Unfortunately, this increased initial working probability could result in nodes becoming **helper** with incorrect estimates of  $n$ , violating the correctness of the termination mechanism. We fix this problem by adding step three to each phase: observing the fraction of silent slots in step three allows nodes to determine the reliability of their estimates.

**Algorithm description.** MULTICASTADVADP contains multiple super-epochs, and the first super-epoch number is  $I_b = 2 \lg C + 20$ . In super-epoch  $i$ , there are  $b_i$  phases numbered from 0 to  $b_i - 1$ , where  $b$  is some large constant. Each phase contains three steps. For any super-epoch  $i$ , the length of each step is always  $R_i = a \cdot 2^i \cdot i^3$ , where  $a$  is some large constant. Prior to execution, all nodes are in **init** status. Similar to MULTICASTADP, each node  $u$  maintains  $M_u$  to indicate whether it knows the message  $m$  or not.

We now describe nodes' behavior in each  $(i, j)$ -phase – i.e., phase  $j$  of super-epoch  $i$  – in detail. For each slot in an  $(i, j)$ -phase, each node will go to a channel chosen uniformly at random. Then, for each node  $u$ , it will broadcast or listen on the chosen channel, each with a certain probability. In step one and two, this probability is  $p_u^{i,j}$ ; in step three, this probability is  $p_{\text{step3}}^i = C^2/2^i$ . We often call  $p_u^{i,j}$  as the working probability of node  $u$ . Notice, at the beginning of an super-epoch  $i$ , the probability  $p_u^{i,j}$ , which is just  $p_u^{i,0}$ , is set to  $C/2^i$ . In a slot, if  $u$  chooses to send, then the broadcast content depends on the value of  $M_u$ : if  $M_u$  is *true* then  $u$  will broadcast  $m$ , otherwise  $u$  will broadcast a beacon message  $\pm$ . On the other hand, if  $u$  chooses to listen in a slot, then it will record the channel feedback. One point worth noting is, a node  $u$  will only change  $M_u$  from *false* to *true* if it hears message  $m$  in step one. (The purpose of this somewhat strange behavior is to facilitate analysis.)

At the end of each phase  $j$ , nodes will compute  $p_u^{i,j+1}$  (i.e., the working probability of the next phase). Specifically, for each node  $u$ , define  $\Delta_u^{\text{step1}} = \Delta_u^{\text{step2}} = R_i p_u^{i,j} / (1 - p_u^{i,j} / C)$  and  $\Delta_u^{\text{step3}} = R_i p_{\text{step3}}^i / (1 - p_{\text{step3}}^i / C)$ . Let  $N_u^{\text{step1},c}$ ,  $N_u^{\text{step2},c}$ , and  $N_u^{\text{step3},c}$  denote the number of

silent slots  $u$  observed in step one, step two, and step three in phase  $j$ , respectively. Then,  $\eta_u^{i,j} = N_u^{step1,c} / \Delta_u^{step1} + N_u^{step2,c} / \Delta_u^{step2} + N_u^{step3,c} / \Delta_u^{step3}$ , and  $p_u^{i,j+1} = p_u^{i,j} \cdot 2^{\max\{0, \eta_u^{i,j} - 2.5\}}$ .

At the end of each phase  $j$ , nodes will also potentially change their status. Specifically, if a node  $u$  is in **init** status and finds: (a)  $\eta_u^{i,j} \geq 2.4$ ; and (b) it has heard the message  $m$  at least  $ai^3$  times during step two of phase  $j$ . Then, node  $u$  will become **helper** and compute an estimate of  $n$  as  $n_u = C / ((p_u^{i,j})^2 \cdot 2^i)$ . On the other hand, if  $u$  is already a **helper** and finds  $p_u^{i,j+1} \geq 64\sqrt{C/(2^i \cdot n_u)}$ , then  $u$  will change its status to **halt** and stop execution. Complete pseudocode of MULTICASTADP is provided in the full version of the paper.

## 6 Analysis of MultiCastAdvAdp

Throughout the analysis, when considering an  $(i, j)$ -phase, we often omit the indices  $i$  and/or  $j$  if they are clear from the context. For any node  $u$ , we often use  $p_u$  to denote its working probability in a step. We always use  $V$  to denote active nodes, and  $M$  to denote active nodes with  $M_u = true$ . Omitted proofs and auxiliary lemmas are provided in the full paper.

**The “bounded difference” property.** The main goal of this part is to show nodes’ working probabilities can never differ too much. This “bounded difference” property is used extensively in remaining analysis, either explicitly or implicitly.

► **Lemma 8.** *Consider a super-epoch  $i > \lg n$ . With probability at least  $1 - \exp(-\Theta(iC))$ , we have  $1/2 \leq p_u/p_v \leq 2$  for any two nodes  $u$  and  $v$  at any phase of the super-epoch.*

At a high level, the above lemma holds because the fraction of silent slots nodes observed during a phase cannot differ too much. To prove it formally, we show the following claim via a coupling argument. However, details of the coupling differ from the ones we saw in Section 4. Specifically, we divide jamming results vectors into  $\binom{R+C}{C}$  categories.

▷ **Claim 9.** Consider a step of length  $R$  and two active nodes  $u$  and  $v$ . Let  $p_u$  (resp.,  $p_v$ ) be the sending/listening probabilities of  $u$  (resp.,  $v$ ); and let  $X_u$  (resp.,  $X_v$ ) be the number of silent slots  $u$  (resp.,  $v$ ) observed. Define  $\Delta_u = Rp_u/(1 - p_u/C)$  and  $\Delta_v = Rp_v/(1 - p_v/C)$ . Define  $\chi_u = \sqrt{giC/(Rp_u)}$  and  $\chi_v = \sqrt{giC/(Rp_v)}$ , where  $g \leq a/20$  is a constant. Then:

1.  $\Pr[X_u/\Delta_u > 1] \leq \exp(-\Theta(i^3C))$ .
2.  $\Pr[(X_u/\Delta_u > 0.2) \wedge (X_v/\Delta_v < 0.1)] \leq \exp(-\Theta(i^3C))$ .
3.  $\Pr[(|X_u/\Delta_u - X_v/\Delta_v| \geq \chi_u + \chi_v) \wedge (X_u/\Delta_u \geq 0.1) \wedge (X_v/\Delta_v \geq 0.1)] \leq \exp(-\Theta(iC))$ .

**Proof sketch.** We begin with part (1). Define  $\alpha = \prod_{w \in V} (1 - p_w/C)$ . To make  $X_u$  as large as possible, assume Eve does no jamming, thus whether  $u$  hears silence are independent among different slots. Notice that  $\mathbb{E}[X_u] = p_u \cdot (\prod_{v \in V \setminus \{u\}} (1 - p_v/C)) \cdot R = \alpha \cdot \Delta_u < \Delta_u$ . Therefore, by a Chernoff bound, the probability that  $X_u > \Delta_u$  is at most  $\exp(-\Theta(\Delta_u)) = \exp(-\Theta(i^3C))$ .

Proofs for part (2) and (3) both rely on coupling, and we only focus on part (2) here.

We first setup the coupling. Assume the randomnesses of nodes come from  $C$  lists  $(T_0, \dots, T_C)$ . Specifically, for each slot  $i$  in the step, if the jamming result is  $Q_i \subseteq [C]$ , then nodes’ behavior in this slot is determined by  $\Psi_{Q_i} \left( T_{|Q_i|}^{\left( \sum_{j \leq i} \mathbb{I}[|Q_j|=|Q_i|] \right)} \right)$  using permutation  $\pi_{Q_i}$  and bijection  $\Psi_{Q_i}$ . Notice,  $\pi_{Q_i}$  and  $\Psi_{Q_i}$  are defined in Section 4 on page 7, and  $T_{|Q_i|}^{\sum_{j \leq i} \mathbb{I}[|Q_j|=|Q_i|]}$  is the  $(\sum_{j \leq i} \mathbb{I}[|Q_j|=|Q_i|])$ -th chunk in list  $T_{|Q_i|}$ . Let  $X_{u,i}$  be an indicator random variable taking value 1 iff  $u$  hears silence in the  $i$ th slot, define  $X_u = \sum_{i=1}^R X_{u,i}$ .

Define  $\mathcal{Z} = \{z = \langle z_1, z_2, \dots, z_C \rangle \in \mathbb{N}^C : \sum_{l=1}^C z_l \leq R\}$ , thus  $|\mathcal{Z}| = \binom{R+C}{C} \leq (R+1)^C \leq (2R)^C$ . (Intuitively, for every  $l \in [C]$ ,  $z_l$  in  $z$  is the number of slots in which Eve leaves



## 22:12 Multi-Channel Resource Competitive Broadcast

$l$  channels unjammed.) Denote the jamming results of this step as  $\mathbf{Q} = (Q_1, \dots, Q_R) \in \mathcal{Q} = (2^{[C]})^R$ , and define  $|\mathbf{Q}| = \sum_{i=1}^R |Q_i|$ . Further define function  $K : \mathcal{Q} \rightarrow \mathcal{Z}$  such that  $K(\mathbf{Q}) = \langle K_1(\mathbf{Q}), \dots, K_C(\mathbf{Q}) \rangle$ , where  $K_l(\mathbf{Q}) = \sum_{i=1}^R \mathbb{I}[|Q_i| = l]$ . (That is,  $K_l(\mathbf{Q})$  counts the number of slots in which Eve leaves  $l$  channels unjammed.) Hence, given  $K(\mathbf{Q})$ , we can use a function  $L : \mathcal{Z} \rightarrow \mathbb{N}$  to compute  $|\mathbf{Q}|$ . In particular,  $L(\mathbf{z}) = \sum_{l=1}^C z_l \cdot l$  and  $L(K(\mathbf{Q})) = |\mathbf{Q}|$ .

Now, consider another execution, for any  $j \geq 1$  and  $l \in [C]$ , let  $Y_{u,l}^{(j)}$  be an indicator random variable taking value 1 iff  $u$  hears silence in a slot in which the jamming result is  $[l]$  and the behavior of nodes is determined by the  $j^{\text{th}}$  chunk of  $\mathbf{T}_l$  directly. Define  $Y_u(\mathbf{z}) = \sum_{l=1}^C \sum_{j=1}^{z_l} Y_{u,l}^{(j)}$  for any  $\mathbf{z} \in \mathcal{Z}$ . By definition, it is easy to verify  $X_u(\mathbf{Q}) = Y_u(K(\mathbf{Q}))$  for any  $\mathbf{Q}$ . That is, for any  $\mathbf{Q}$ , values of  $X_u$  and  $Y_u$  are identical. The significance of this observation is that it relates  $X_u$  – which counts the number of silent slots  $u$  heard – to  $Y_u$ , and  $Y_u$  can be interpreted as the sum of independent random variables once  $\mathbf{z}$  is fixed.

Now we are ready to prove part (2). Notice  $\mathbb{E}[X_u]/\Delta_u = \mathbb{E}[X_v]/\Delta_v = \alpha \cdot |\mathbf{Q}|/(RC)$ . Also, it is easy to verify  $\mathbb{E}[Y_u(\mathbf{z})]/\Delta_u = \mathbb{E}[Y_v(\mathbf{z})]/\Delta_v = \alpha \cdot L(\mathbf{z})/(RC)$ . Let  $\mathcal{Z}_1 = \{\mathbf{z} \in \mathcal{Z} : L(\mathbf{z}) \leq 0.15RC/\alpha\}$ . Then for  $\mathbf{z} \in \mathcal{Z}_1$ ,  $\mathbb{E}[Y_u(\mathbf{z})] \leq 0.15\Delta_u$ , further by a Chernoff bound,  $\Pr[Y_u(\mathbf{z}) > 0.2\Delta_u] \leq \exp(-\Theta(i^3C))$ . Similarly, for  $\mathbf{z} \in \mathcal{Z} \setminus \mathcal{Z}_1$ ,  $\Pr[Y_v(\mathbf{z}) < 0.1\Delta_v] \leq \exp(-\Theta(i^3C))$ . Therefore, we can conclude  $\Pr[X_u(\mathbf{Q}) > 0.2\Delta_u \wedge X_v(\mathbf{Q}) < 0.1\Delta_v] \leq (\sum_{\mathbf{z} \in \mathcal{Z}_1} \Pr[Y_u(\mathbf{z}) > 0.2\Delta_u]) + (\sum_{\mathbf{z} \in \mathcal{Z} \setminus \mathcal{Z}_1} \Pr[Y_v(\mathbf{z}) < 0.1\Delta_v]) \leq |\mathcal{Z}| \cdot \exp(-\Theta(i^3C)) = \exp(-\Theta(i^3C))$ .  $\blacktriangleleft$

We now sketch the proof of Lemma 8. Denote the working probabilities of the current phase and the next phase as  $p$  and  $p'$ . If  $\eta_u \leq 2.5$  and  $\eta_v \leq 2.5$ , then  $p'_u/p'_v = p_u/p_v$  and we are done. So assume  $\eta_u > 2.5$ . In such case, Claim 9 imply  $|N_u^{c,step*}/\Delta_u^{step*} - N_v^{c,step*}/\Delta_v^{step*}| \leq \sqrt{giC/(Rp_u)} + \sqrt{giC/(Rp_v)}$  for any step  $*$  in  $\{1, 2\}$ , and  $|N_u^{c,step3}/\Delta_u^{step3} - N_v^{c,step3}/\Delta_v^{step3}| \leq 2\sqrt{giC/(Rp_{step3})}$ . This further suggests  $p'_u/p'_v \leq (p_u/p_v) \cdot 2^{1/bi}$ , thus the lemma is proved.

**Correctness.** This part shows MULTICASTADVADP enforces two nice properties. First, when some node halts, all nodes must have become **helper**. This property can be seen as a stronger version of Lemma 7, since a node must have heard the message  $m$  when becoming a **helper**. The second property, on the other hand, states that when a node becomes **helper**, it also obtains a good estimate of  $n$ . This property helps to ensure nodes can stop execution at the right time.

► **Lemma 10** (“halt-imply-helper” property). *The probability that some node has stopped execution while some other node has not become **helper** is at most  $n^{-\Omega(1)}$ .*

► **Lemma 11** (“good-estimate” property). *For each node  $u$ , the probability that  $u$  becomes **helper** with  $n_u < n/256$  or  $n_u > 4n$  is at most  $n^{-\Omega(1)}$ .*

The following lemma is helpful for proving both of the above two properties. Roughly speaking, this lemma states that if in an  $(i, j)$ -phase some node  $u$  has working probability  $p_u = \Theta(\sqrt{C/(2^i n)})$  and decides to raise  $p_u$  at the end of the phase, then all nodes must have heard the message many times in step two of the phase.

► **Lemma 12.** *Consider an  $(i, j)$ -phase where  $i > \lg n$ . Assume at the beginning of the phase:  $(\sum_{u \in V} p_u)/C \leq 1/2$ , all nodes are active and their working probabilities are within a factor of two, and the working probability of each node is at least  $8\sqrt{C/(2^i n)}$ . Then, with probability at most  $\exp(-\Theta(i^2))$ , these two events both occur: (a) some node raises its working probability at the end of the phase; and (b) some node hears message  $m$  less than  $ai^3$  times in step two.*



**Proof sketch.** Let  $\mathcal{E}_R$  be the event that some node raises its working probability at the end of the phase,  $\mathcal{E}_M$  be the event that some node hears  $m$  less than  $ai^3$  times during step two,  $\mathcal{E}_{un}$  be the event that some node is still uninformed by the end of step one. Moreover, let  $\mathcal{E}_1$  (respectively,  $\mathcal{E}_2$ ) be the event that  $\mathcal{E}_{step1}^{\geq 0.25}$  ( $\geq 0.25$ ) (respectively,  $\mathcal{E}_{step2}^{\geq 0.25}$  ( $\geq 0.25$ )) occurs during step one (respectively, step two) of the phase. We know:

$$\begin{aligned} \Pr(\mathcal{E}_M \mathcal{E}_R) &\leq \Pr(\mathcal{E}_M \wedge (\mathcal{E}_1 \wedge \mathcal{E}_2)) + \Pr(\mathcal{E}_R \wedge \overline{(\mathcal{E}_1 \wedge \mathcal{E}_2)}) \\ &\leq \Pr(\mathcal{E}_{un} \mathcal{E}_1) + \Pr(\overline{\mathcal{E}_{un}} \mathcal{E}_M \mathcal{E}_2) + \Pr(\mathcal{E}_R \wedge (\overline{\mathcal{E}_1} \vee \overline{\mathcal{E}_2})) \end{aligned}$$

The remainder of the proof bounds the three probabilities in the last line.

*Claim I:*  $\Pr(\mathcal{E}_{un} \mathcal{E}_1) \leq \Pr(\mathcal{E}_{un} | \mathcal{E}_1) \leq \exp(-\Theta(i^2))$ . If  $\mathcal{E}_1$  happens, then step one is not heavily jammed. Thus every node will be informed at the end of step one due to the effectiveness of the epidemic broadcast scheme, much like the proof of Lemma 5.

*Claim II:*  $\Pr(\overline{\mathcal{E}_{un}} \mathcal{E}_M \mathcal{E}_2) \leq \Pr(\mathcal{E}_M \mathcal{E}_2 | \overline{\mathcal{E}_{un}}) \leq \exp(-\Theta(i^3))$ . Fix a node  $u$ , and assume all nodes know  $m$  at the beginning of step two. Similar to the proof of Lemma 6 (except that we focus on message slots and apply the coupling argument accordingly), the probability that  $u$  hears  $m$  less than  $ai^3$  times during a step two in which  $\mathcal{E}_2$  occurs is at most  $\exp(-\Theta(i^3))$ . Take a union over all nodes and the claim is proved.

*Claim III:*  $\Pr(\mathcal{E}_R \wedge (\overline{\mathcal{E}_1} \vee \overline{\mathcal{E}_2})) \leq \exp(-\Theta(i^3 C))$ . Notice that  $\Pr(\mathcal{E}_R \wedge (\overline{\mathcal{E}_1} \vee \overline{\mathcal{E}_2})) \leq \Pr(\mathcal{E}_R \overline{\mathcal{E}_1}) + \Pr(\mathcal{E}_R \overline{\mathcal{E}_2}) \leq \sum_{u \in V} \Pr(\mathcal{E}_{u,1} \overline{\mathcal{E}_1}) + \sum_{u \in V} \Pr(\mathcal{E}_{u,2} \overline{\mathcal{E}_2}) + 4 \sum_{u \in V} \exp(-\Theta(i^3 C))$ . Here,  $\mathcal{E}_{u,1}$  (respectively,  $\mathcal{E}_{u,2}$ ) is the event that node  $u$  hears silence more than  $\Delta_u^{step1}/2$  (respectively,  $\Delta_u^{step2}/2$ ) times in step one (respectively, step two) of the phase, and the last inequality is due to part (1) of Claim 9. When  $\overline{\mathcal{E}_1}$  occurs, the expected number of silent slots heard by  $u$  in step one is at most  $7/16 \Delta_u^{step1}$ . Again via a coupling argument, we know  $\Pr(\mathcal{E}_{u,1} \overline{\mathcal{E}_1}) \leq \exp(-\Theta(i^3))$ , and bounding  $\Pr(\mathcal{E}_{u,2} \overline{\mathcal{E}_2})$  is similar.  $\blacktriangleleft$

At this point, to prove the “halt-imply-helper” property, we only need to combine the above lemma with the following two observations. First, nodes are unlikely to become **helper** in early super-epochs, as the sending probabilities in these super-epochs are too high and nodes cannot hear enough silent slots. Second, when nodes’ working probabilities in step two are too small, they will also not become **helper** as the number of messages heard is not enough. Notice, this second observation also leads to an upper bound on the estimates of  $n$ . (Detailed proofs of the two observations can be found in the full paper.)

To prove the “good-estimate” property, what remains is to show a lower bound for  $n_u$ . To that end, we show if all nodes are alive and  $u$ ’s working probability is close to the ideal value  $\Theta(\sqrt{C}/(2^i n))$ , then  $u$  must have become **helper** already. (Again, see the full paper for the proof.) By then, a lower bound of  $n_u$  can be derived as a simple corollary of this claim.

**Termination.** This part shows nodes will quickly become **helper** and then halt once jamming is weak. (In other words, Eve cannot delay nodes unless she spends a lot of energy.) We begin by classifying phases and super-epochs into *weakly jammed* ones and *strongly jammed* ones. Specifically, call a phase weakly jammed if  $\mathcal{E}^{\geq 0.95}$  ( $\geq 0.95$ ) occurs for all three steps of the phase. Otherwise, if  $\mathcal{E}^{> 0.95}$  ( $< 0.95$ ) occurs for any of the three steps, then the phase is strongly jammed. Call a super-epoch weakly jammed if at least half of the phases in the super-epoch are weakly jammed, otherwise the super-epoch is strongly jammed.

We first show, if a node’s working probability has not reached the ideal value, then this probability will increase by some constant factor in a weakly jammed phase.

$\blacktriangleright$  **Lemma 13.** Fix an  $(i, j)$ -phase where  $i \geq \lg(nC) + 6$ , and fix an active node  $u$  satisfying  $p_u^{i,j} < C/(128n)$ . By the end of the phase, the following two events happen simultaneously with probability  $\exp(-\Omega(iC))$ : (a) the phase is weakly jammed; and (b)  $p_u^{i,j+1} < p_u^{i,j} \cdot 2^{(1/10)}$ .

Building upon Lemma 13, we can prove nodes' working probabilities will reach  $\tilde{p}_i = 1024\sqrt{C/(2^i n)}$  in a weakly jammed super-epoch, as there are enough weakly jammed phases.

► **Lemma 14.** *Fix a super-epoch  $i \geq 34 + \lg(nC)$  and a node  $u$  that is active at the beginning of the super-epoch. The following two events happen simultaneously with probability  $\exp(-\Omega(iC))$ : (a) the super-epoch is weakly jammed; and (b) by the end of the super-epoch  $u$  is still alive with a working probability less than  $\tilde{p}_i$ .*

Lastly, we show that when a node's working probability reaches  $\tilde{p}_i$ , it will halt.

► **Lemma 15.** *Fix a super-epoch  $i \geq \lg(nC) - 7$  and a node  $u$ . Assume the "halt-imply-helper" property and the "good-estimate" property both hold. Then, the probability that  $u$  is active at the end of super-epoch  $i$  with a working probability exceeding  $\tilde{p}_i$  is at most  $\exp(-\Theta(i))$ .*

**Main theorem.** In this last part we sketch the proof of Theorem 2.

Fix an arbitrary node  $u$ . The first step is to analyze how long  $u$  remains active. Since super-epoch length increases geometrically, we only need to focus on the last super-epoch in which  $u$  is active. Specifically, let  $\hat{I} = 34 + \lg C + \max\{\lg C, \lg n\}$ , let  $r_i$  be the number of slots in super-epoch  $i$ , and let  $sr_i = \sum_{k=\hat{I}+1}^i r_k$  be the total number of slots from super-epoch  $\hat{I} + 1$  to super-epoch  $i$ . It is easy to verify, for  $i \geq \hat{I} + 1$ ,  $sr_i \leq 5r_{i-1}$ . Define constant  $\beta = 2400$ , and let random variable  $L$  denote node  $u$ 's actual runtime starting from super-epoch  $\hat{I} + 1$ . Combine Lemma 10, 11, 14, 15, along with the fact that Eve spends less than  $r_i C / \beta = bi/2 \cdot 0.05^2 R_i C$  energy in super-epoch  $i$  implies super-epoch  $i$  is weakly jammed, we can prove  $L \leq 5\beta T / C$  holds w.h.p. Take a union bound over all nodes, we know every node will terminate within  $(\sum_{k=I_b}^{\hat{I}} bk \cdot 3R_k) + 5\beta T / C = \tilde{O}(T/C + nC + C^2)$  slots, w.h.p.

Next, we analyze the cost of node  $u$ . Let  $F_{step1,2}$  (resp.,  $F_{step3}$ ) be node  $u$ 's total actual cost during step one and step two (resp., step three) in all phases starting from super-epoch  $\hat{I} + 1$ . By an analysis similar to above, we show  $F_{step1,2} \leq \Theta(\sqrt{T/n} \cdot \lg^2 T)$  and  $F_{step3} \leq \Theta(C^2 \cdot (\hat{I} + \lg T)^5)$ , w.h.p. As a result, we can conclude w.h.p. the energy cost of each node is bounded by  $F_{step1,2} + F_{step3} + \sum_{k=I_b}^{\hat{I}} (bk \cdot 3R_k) = \tilde{O}(\sqrt{T/n} + nC + C^2)$ .

Finally, notice the algorithm itself ensures a node must be informed when it halts.

## 7 Lower Bounds

In this section, we show our algorithms achieve (near) optimal time and energy complexity simultaneously against an adaptive adversary with budget  $T$ . The time complexity part is obvious: Eve can jam all channels during the first  $T/C$  slots, so the  $O(T/C)$  term in the runtime of MULTICASTADP and MULTICASTADVADP is asymptotically optimal.

Obtaining an energy complexity lower bound is much more involved. To do so, the first step is a simulation argument. Specifically, given any fair multi-channel broadcast algorithm  $\mathcal{A}_n$ , we can devise a multi-channel 1-to-1 communication algorithm  $\mathcal{A}_2$  (in which the goal is to let one node called Alice to send a message  $m$  to another node Bob) that simulates  $\mathcal{A}_n$  internally. To make the simulation feasible, we allow Alice and Bob to have multiple transceivers, so that in each slot they can operate on multiple channels, as well as send and listen simultaneously. In more detail, Alice in  $\mathcal{A}_2$  mimics the source node in  $\mathcal{A}_n$ . As for Bob, he simulates the  $n - 1$  non-source nodes in  $\mathcal{A}_n$ . Particularly, in each slot, for each channel, if at least one non-source node listens, then Bob uses a transceiver to listen; if exactly one non-source node broadcasts, then Bob uses a transceiver to broadcast the unique message; and if at least two non-source nodes broadcast, then Bob uses a transceiver to broadcast noise. (Notice Bob can simultaneously listen and broadcast on a channel: he uses two transceivers

and incurs two units of cost.) On the other hand, Eve’s strategy for disrupting  $\mathcal{A}_n$  and  $\mathcal{A}_2$  is called  $\mathcal{S}$ : in each slot, for each channel, Eve jams it iff the probability that the source node (respectively, Alice) successfully transmits  $m$  to some non-source node (respectively, Bob) over this channel exceeds  $1/T$ .

Clearly, the above simulation is “perfect”: an execution of  $\mathcal{A}_2$  is identical to an execution of  $\mathcal{A}_n$ , assuming nodes use identical random bits in the two executions. To simplify presentation, we further assume  $\mathcal{A}_n$  automatically stops once all nodes are informed, and  $\mathcal{A}_2$  automatically stops once Bob is informed. This modification will not increase nodes’ energy cost, thus will not affect the correctness of our lower bound. Now, observe that the success of  $\mathcal{A}_2$  is a necessary condition for the success of  $\mathcal{A}_n$ , and Bob’s energy cost will not exceed the sum of all non-source nodes’ cost, hence the following lemma is immediate.

► **Lemma 16.** *For any fair multi-channel broadcast algorithm  $\mathcal{A}_n$ , there exists a multi-channel 1-to-1 communication algorithm  $\mathcal{A}_2$ . If in  $\mathcal{A}_n$  each node incurs an expected cost of  $f(T)$  and  $\mathcal{A}_n$  succeeds with probability  $p$ , then: (a) in  $\mathcal{A}_2$  Alice and Bob incur an expected cost of at most  $f(T)$  and  $n \cdot f(T)$ , respectively; (b)  $\mathcal{A}_2$  succeeds with probability at least  $p$ .*

What remains is an energy complexity lower bound for  $\mathcal{A}_2$ : with such a result, Theorem 3 is immediate via simple reduction. Indeed, we are able to prove Theorem 17, an energy complexity lower bound for 1-to-1 communication in the multi-channel setting. This result could be of independent interest, and at a high-level its proof is organized in the following way. First, we note that in a rough sense, any multi-channel 1-to-1 communication algorithm  $\mathcal{A}$  can be viewed as a decision tree, and each path from the root to a leaf in the tree corresponds to an oblivious algorithm. Then, we argue that  $\mathcal{A}$  can be used to generate another algorithm  $\mathcal{A}'$  which is a “convex combination” (or, a distribution) of all such oblivious algorithms, without changing the success probability or the product of Alice’s and Bob’s expected cost. Moreover, an important observation is that among all the oblivious algorithm used in the “convex combination”, at least one – say  $\mathcal{A}_{\hat{w}}$  – is (roughly) as good as  $\mathcal{A}'$  in terms of both success probability and energy efficiency. Finally, depending on whether Eve uses all her budget during execution, we consider two potential scenarios for  $\mathcal{A}_{\hat{w}}$ , and for both we show  $\mathbb{E}_{\mathcal{A}_{\hat{w}}}[A] \cdot \mathbb{E}_{\mathcal{A}_{\hat{w}}}[B] \in \Omega(T)$ , which in turn implies  $\mathbb{E}_{\mathcal{A}}[A] \cdot \mathbb{E}_{\mathcal{A}}[B] \in \Omega(T)$ . Complete proof of Theorem 17 is provided in the full paper.

► **Theorem 17.** *Consider any multi-channel 1-to-1 communication algorithm that succeeds with constant probability against an adaptive adversary Eve with budget  $T$ . Let  $A$  and  $B$  denote Alice’s and Bob’s expected cost respectively, then Eve can force  $\mathbb{E}[A] \cdot \mathbb{E}[B] \in \Omega(T)$ .*

---

## References

- 1 John Augustine, Valerie King, Anisur Molla, Gopal Pandurangan, and Jared Saia. Scalable and secure computation among strangers: Message-competitive byzantine protocols. In *International Symposium on Distributed Computing*, DISC '20. Springer, 2020.
- 2 Baruch Awerbuch, Andrea Richa, and Christian Scheideler. A jamming-resistant mac protocol for single-hop wireless networks. In *Proceedings of the 27th ACM Symposium on Principles of Distributed Computing*, PODC '08, pages 45–54. ACM, 2008.
- 3 Reuven Bar-Yehuda, Oded Goldreich, and Alon Itai. On the time-complexity of broadcast in multi-hop radio networks: An exponential gap between determinism and randomization. *Journal of Computer and System Sciences*, 45(1):104–126, 1992.
- 4 M. Bender, J. Fineman, M. Movahedi, J. Saia, V. Dani, S. Gilbert, S. Pettie, and M. Young. Resource-competitive algorithms. *SIGACT News*, 46(3):57–71, 2015.

- 5 Yi-Jun Chang, Varsha Dani, Thomas Hayes, Qizheng He, Wenzheng Li, and Seth Pettie. The energy complexity of broadcast. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, PODC '18, pages 95–104. ACM, 2018.
- 6 Yi-Jun Chang, Varsha Dani, Thomas P. Hayes, and Seth Pettie. The energy complexity of bfs in radio networks. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, PODC '20, pages 27–282. ACM, 2020.
- 7 Yi-Jun Chang, Tsvi Kopelowitz, Seth Pettie, Ruosong Wang, and Wei Zhan. Exponential separations in the energy complexity of leader election. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, STOC '17, pages 771–783. ACM, 2017.
- 8 Haimin Chen and Chaodong Zheng. Fast and resource competitive broadcast in multi-channel radio networks. In *Proceedings of the 31st ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '19, pages 179–189. ACM, 2019.
- 9 Haimin Chen and Chaodong Zheng. Broadcasting competitively against adaptive adversary in multi-channel radio networks. arXiv, 2020. URL: <https://arxiv.org/abs/2001.03936>.
- 10 Artur Czumaj and Peter Davies. Exploiting spontaneous transmissions for broadcasting and leader election in radio networks. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC '17, pages 3–12. ACM, 2017.
- 11 Shlomi Dolev, Seth Gilbert, Rachid Guerraoui, and Calvin Newport. Gossiping in a multi-channel radio network. In *International Symposium on Distributed Computing*, DISC '07, pages 208–222. Springer Berlin Heidelberg, 2007.
- 12 Devdatt Dubhashi and Alessandro Panconesi. *Concentration of Measure for the Analysis of Randomized Algorithms*. Cambridge University Press, 2009.
- 13 Leszek Gasieniec, Erez Kantor, Dariusz R. Kowalski, David Peleg, and Chang Su. Energy and time efficient broadcasting in known topology radio networks. In *International Symposium on Distributed Computing*, DISC '07, pages 253–267. Springer Berlin Heidelberg, 2007.
- 14 Mohsen Ghaffari, Bernhard Haeupler, and Majid Khabbazi. Randomized broadcast in radio networks with collision detection. *Distributed Computing*, 28(6):407–422, 2015.
- 15 Seth Gilbert, Valerie King, Seth Pettie, Ely Porat, Jared Saia, and Maxwell Young. (near) optimal resource-competitive broadcast with jamming. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, pages 257–266. ACM, 2014.
- 16 Ramakrishna Gummadi, David Wetherall, Ben Greenstein, and Srinivasan Seshan. Understanding and mitigating the impact of rf interference on 802.11 networks. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '07, pages 385–396. ACM, 2007.
- 17 Marcin Kardas, Marek Klonowski, and Dominik Pajak. Energy-efficient leader election protocols for single-hop radio networks. In *2013 42nd International Conference on Parallel Processing*, ICPP '13, pages 399–408. IEEE, 2013.
- 18 Valerie King, Seth Pettie, Jared Saia, and Maxwell Young. A resource-competitive jamming defense. *Distributed Computing*, 31(6):419–439, 2018.
- 19 Valerie King, Jared Saia, and Maxwell Young. Conflict on a communication channel. In *Proceedings of the 30th ACM Symposium on Principles of Distributed Computing*, PODC '11, pages 277–286. ACM, 2011.
- 20 Dominic Meier, Yvonne Anne Pignolet, Stefan Schmid, and Roger Wattenhofer. Speed dating despite jammers. In *International Conference on Distributed Computing in Sensor Systems*, DCOSS '09, pages 1–14. Springer Berlin Heidelberg, 2009.
- 21 Joseph Polastre, Robert Szewczyk, and David Culler. Telos: enabling ultra-low power wireless research. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*, IPSN '05, pages 364–369. IEEE, 2005.
- 22 Andrea Richa, Christian Scheideler, Stefan Schmid, and Jin Zhang. A jamming-resistant mac protocol for multi-hop wireless networks. In *International Symposium on Distributed Computing*, DISC '10, pages 179–193. Springer Berlin Heidelberg, 2010.

# Dynamic Byzantine Reliable Broadcast

**Rachid Guerraoui**

EPFL, Lausanne, Switzerland

**Jovan Komatovic**

EPFL, Lausanne, Switzerland

**Petr Kuznetsov**

LTCI, Télécom Paris

Institut Polytechnique Paris, France

**Yvonne-Anne Pignolet**

DFINITY Foundation, Zürich, Switzerland

**Dragos-Adrian Seredinschi**

Informal Systems, Lausanne, Switzerland

**Andrei Tonkikh**

National Research University Higher School of Economics, St. Petersburg, Russia

---

## Abstract

---

Reliable broadcast is a communication primitive guaranteeing, intuitively, that all processes in a distributed system deliver the same set of messages. The reason why this primitive is appealing is twofold: (i) we can implement it deterministically in a completely asynchronous environment, unlike stronger primitives like consensus and total-order broadcast, and yet (ii) reliable broadcast is powerful enough to implement important applications like payment systems.

The problem we tackle in this paper is that of *dynamic* reliable broadcast, i.e., enabling processes to join or leave the system. This property is desirable for long-lived applications (aiming to be highly available), yet has been precluded in previous asynchronous reliable broadcast protocols. We study this property in a general adversarial (i.e., Byzantine) environment.

We introduce the first specification of a dynamic Byzantine reliable broadcast (DBRB) primitive that is amenable to an asynchronous implementation. We then present an algorithm implementing this specification in an asynchronous network. Our DBRB algorithm ensures that if any correct process in the system broadcasts a message, then every correct process delivers that message unless it leaves the system. Moreover, if a correct process delivers a message, then every correct process that has not expressed its will to leave the system delivers that message. We assume that more than  $2/3$  of processes in the system are correct at all times, which is tight in our context.

We also show that if only one process in the system can fail – and it can fail only by crashing – then it is impossible to implement a stronger primitive, ensuring that if any correct process in the system broadcasts or delivers a message, then every correct process in the system delivers that message – including those that leave.

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

**Keywords and phrases** Byzantine reliable broadcast, deterministic distributed algorithms, dynamic distributed systems

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2020.23

**Related Version** A full version of the paper is available at <https://arxiv.org/abs/2001.06271>.

**Funding** This Work Has Been Supported in Part by the Interchain Foundation, Cross-Chain Validation Project.



© Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, and Andrei Tonkikh;  
licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 23; pp. 23:1–23:18



Leibniz International Proceedings in Informatics

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

## 1 Introduction

Networks typically offer a reliable form of communication channels: TCP. As an abstraction, these channels ensure that if neither the sender nor the destination of a message fail, then the message is eventually delivered. Essentially, this abstraction hides the unreliability of the underlying IP layer, so the user of a TCP channel is unaware of the lost messages.

Yet, for many applications, TCP is not reliable enough. Indeed, think of the situation where a message needs to be sent to all processes of a distributed system. If the sender does not fail, TCP will do the job; but otherwise, the message might reach only a strict subset of processes. This can be problematic for certain applications, such as a financial notification service when processes subscribe to information published by other processes. For fairness reasons, one might want to ensure that if the sender fails, either *all or no process* delivers that message. Moreover, if the correct processes choose to deliver, they must deliver the same message, even when the sender is Byzantine. We talk, therefore, about *reliable broadcast*. Such a primitive does not ensure that messages are delivered in the same total order, but simply in the “all-or-nothing” manner.

Reliable broadcast is handy for many applications, including, for example, cryptocurrencies. Indeed, in contrast to what was implicitly considered since Nakamoto’s original paper [24], there is no need to ensure consensus on the ordering of messages, i.e., to totally order messages, if the goal is to perform secure payments. A reliable broadcast scheme suffices [15].

Reliable broadcast is also attractive because, unlike stronger primitives such as total order broadcast and consensus, it can be implemented deterministically in a completely asynchronous environment [7]. The basic idea uses a quorum of correct processes, and makes that quorum responsible for ensuring that a message is transmitted to all processes if the original sender of the message fails. If a message does not reach the quorum, it will not be delivered by any process. It is important to notice at this point a terminology difference between the act of “receiving” and the act of “delivering” a message. A process indeed might “receive” a message  $m$ , but not necessarily “deliver”  $m$  to its application until it is confident that the “all-or-nothing” property of the reliable broadcast is ensured.

A closer look at prior asynchronous implementations of reliable broadcast reveals, however, a gap between theory and practice. The implementations described so far all assume a *static* system. Essentially, the set of processes in the system remains the same, except that some of them might fail. The ability of a process to join or leave the system, which is very desirable in a long-lived application supposed to be highly available, is precluded in all asynchronous reliable broadcast protocols published so far.

In this paper, we introduce the first specification of a *dynamic* Byzantine reliable broadcast (DBRB) primitive that is amenable to an asynchronous implementation. The specification allows any process outside the broadcast system to join; any process that is inside the system can ask to leave. Processes inside the system can broadcast and deliver messages, whereas processes outside the system cannot. Our specification is intended for an asynchronous system for it does not require the processes to agree on the system membership. Therefore, our specification does not build on top of a group membership scheme, as does the classical *view synchrony* abstraction [10].

Our asynchronous DBRB implementation ensures that if any correct process in the system broadcasts a message, then eventually every correct process, unless it asks to leave the system, delivers that message. Moreover, if any correct process delivers a message, then every correct process, if it has not asked to leave prior to the delivery, delivers that message. The main technical difficulty addressed by our algorithm is to combine asynchrony and dynamic membership, which makes it impossible for processes to agree on the exact membership.



Two key insights enable us to face this challenge. First, starting from a known membership set at system bootstrap time, we construct a sequence of changes to this set; at any time, there is a majority of processes that record these changes. Based on this sequence, processes can determine the validity of messages. Second, before transitioning to a new membership, correct processes exchange their current state with respect to “in-flight” broadcast messages and membership changes. This prevents equivocation and conflicts.

Our algorithm assumes that, at any point in time, more than  $2/3$  of the processes inside the broadcast system are correct, which is tight. Moreover, we show that the “all-or-nothing” property we ensure is, in some sense, maximal. More precisely, we prove (see [14]) that in an asynchronous system, even if only one process in the system can fail, and it can merely fail by crashing, then it is impossible to implement a stronger property, ensuring that if any correct process in the system broadcasts (resp., delivers) a message, then every correct process in the system delivers that message, including those that are willing to leave.

The paper is organized as follows. In §2, we describe our system model and introduce the specification of DBRB. In §3, we overview the structure of our algorithm. In §4, we describe our implementation, and in §5, we argue its correctness. We conclude in §6 with a discussion of related and future work. Detailed proofs are delegated to the full version of the paper [14].

## 2 Model and Specification

We describe here our system model (§2.1) and specify our DBRB primitive (§§ 2.2 to 2.4).

### 2.1 A Universe of Asynchronous Processes

We consider a universe  $\mathcal{U}$  of processes, subject to *Byzantine* failures: a faulty process may arbitrarily deviate from the algorithm it is assigned. Processes that are not subject to failures are *correct*. We assume an asymmetric cryptographic system. Correct processes communicate with signed messages: prior to sending a message  $m$  to a process  $q$ , a process  $p$  signs  $m$ , labeled  $\langle m \rangle_{\sigma_p}$ . Upon receiving the message,  $q$  can verify its authenticity and use it to prove its origin to others (non-repudiation). To simplify presentation, we omit the signature-related notation and, thus, whenever we write  $m$ , the identity of sender  $p$  and the signature are implicit and correct processes only consider messages, whether received directly or relayed by other processes, if they are equipped with valid signatures. We also use the terms “send” and “disseminate” to differentiate the points in our algorithm when a process sends a message, resp., to a single or to many destinations.

The system  $\mathcal{U}$  is *asynchronous*: we make no assumptions on communication delays or relative speeds of the processes. We assume that communication is *reliable*, i.e., every message sent by a correct process to a correct process is eventually received. To describe the events that occur to an external observer and prove the correctness of the protocol, we assume a global notion of time, outside the control of the processes (not used in the protocol implementation). We consider a subset of  $\mathcal{U}$  called the *broadcast system*. We discuss below how processes join or leave the broadcast system.

### 2.2 DBRB Interface

Our DBRB primitive exposes an interface with three operations and one callback:

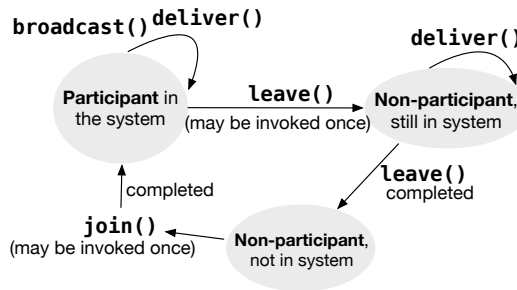
1. DBRB-JOIN: used by a process outside the system to *join*.
2. DBRB-LEAVE: used by a process inside the system to *leave*.



- 3. DBRB-BROADCAST( $m$ ): used by a process inside the system to broadcast a message  $m$ .
- 4. DBRB-DELIVER( $m$ ): this callback is triggered to handle the delivery of a message  $m$ .

If a process is in the system initially, or if it has returned from the invocation of a DBRB-JOIN call, we say that it has *joined* the system. Furthermore, it is considered *participating* (or, simply, a *participant*) if it has not yet invoked DBRB-LEAVE. When the invocation of DBRB-LEAVE returns, we say that the process *leaves* the system. Note that in the interval between the invocation and the response of a DBRB-LEAVE call, the process is no longer participating, but has not yet left the system.

The following rules (illustrated in Figure 1) govern the behavior of correct processes: (i) a DBRB-JOIN operation can only be invoked if the process is not participating; moreover, we assume that DBRB-JOIN is invoked at most once; (ii) only a participating process can invoke a DBRB-BROADCAST( $m$ ) operation; (iii) a DBRB-DELIVER( $m$ ) callback can be triggered only if a process has previously joined but has not yet left the system; (iv) a DBRB-LEAVE operation can only be invoked by a participating process.



■ **Figure 1** State transition diagram for correct processes.

### 2.3 Standard Assumptions

We make two standard assumptions in asynchronous reconfiguration protocols [1, 2, 5, 26], which we restate below for the sake of completeness.

► **Assumption 1** (Finite number of reconfiguration requests). *In every execution, the number of processes that want to join or leave the system is finite.*

► **Assumption 2**. *Initially, at time 0, the set of participants is nonempty and known to every process in  $\mathcal{U}$ .*

Assumption 1 captures the assumption that no new reconfiguration requests will be made for “sufficiently long”, thus ensuring that started operations do complete. Assumption 2 is necessary to bootstrap the system and guarantees that all processes have the same starting conditions. Additionally, we make standard cryptographic assumptions regarding the power of the adversary, namely that it cannot subvert cryptographic primitives, e.g., forge a signature.

We also assume that a weak broadcast primitive is available. The primitive guarantees that if a correct process broadcasts a message  $m$ , then every correct process eventually delivers  $m$ . In practice, such primitive can be implemented by some sort of a gossip protocol [18]. This primitive is “global” in a sense that it does not require a correct process to know all the members of  $\mathcal{U}$ .

## 2.4 Properties of DBRB

For simplicity of presentation, we assume a specific instance of DBRB in which a predefined sender process  $s$  disseminates a single message via DBRB-BROADCAST operation. The specification can easily be extended to the general case in which every participant can broadcast multiple messages, assuming that every message is uniquely identified.

► **Definition 1** (DBRB basic guarantees).

- **Validity.** *If a correct participant  $s$  broadcasts a message  $m$  at time  $t$ , then every correct process, if it is a participant at time  $t' \geq t$  and never leaves the system, eventually delivers  $m$ .*
- **Totality.** *If a correct process  $p$  delivers a message  $m$  at time  $t$ , then every correct process, if it is a participant at time  $t' \geq t$ , eventually delivers  $m$ .*
- **No duplication.** *A message is delivered by a correct process at most once.*
- **Integrity.** *If some correct process delivers a message  $m$  with sender  $s$  and  $s$  is correct, then  $s$  previously broadcast  $m$ .<sup>1</sup>*
- **Consistency.** *If some correct process delivers a message  $m$  and another correct process delivers a message  $m'$ , then  $m = m'$ .*
- **Liveness.** *Every operation invoked by a correct process eventually completes.*

To filter out implementations that involve *all* processes in the broadcast protocol, we add the following non-triviality property.

► **Definition 2** (Non-triviality). *No correct process sends any message before invoking DBRB-JOIN or after returning from DBRB-LEAVE operation.*

## 3 Overview

We now present the building blocks underlying our DBRB algorithm (§3.1) and describe typical scenarios: (1) a correct process joining or leaving the system (§3.2), and (2) a broadcast (§3.3).

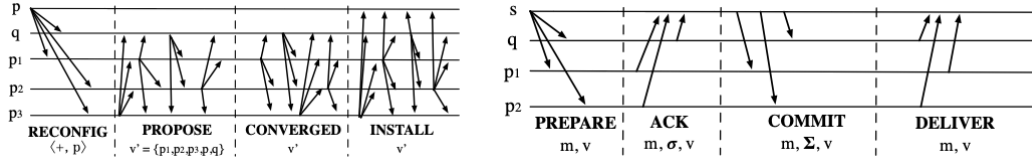
### 3.1 Building Blocks

**Change.** We define a set of *system updates*  $change = \{+, -\} \times \mathcal{U}$ , where the tuple  $\langle +, p \rangle$  (resp.,  $\langle -, p \rangle$ ) indicates that process  $p$  asked to join (resp., leave) the system. This abstraction captures the evolution of system membership throughout time. It is inevitable that, due to asynchrony, processes might not be able to agree on an unique system membership. In other words, two processes may concurrently consider different sets of system participants to be valid. To capture this divergence, we introduce the *view* abstraction, which defines the system membership through the lenses of some specific process at a specific point in time.

**View.** A view  $v$  comprises a set of updates  $v.changes$ . The set determines the view *membership* as  $v.members = \{p \in \mathcal{U} : \langle +, p \rangle \in v.changes \wedge \langle -, p \rangle \notin v.changes\}$ . For simplicity, sometimes we use  $p \in v$  instead of  $p \in v.members$ ;  $|v|$  is a shorthand for  $|v.members|$ .

Intuitively, each correct process  $p$  in DBRB uses a view as an append-only set, to record all the *changes* that the broadcast system underwent up to a point in time, as far as  $p$  observed. Some views are “instantiated” in DBRB protocol and those views are marked as *valid* (a

<sup>1</sup> Recall that the identity of sender process  $s$  for a given message  $m$  is implicit in the message (§2.1).



■ **Figure 2** Protocol overview for DBRB-JOIN or DBRB-LEAVE (left), and DBRB-BROADCAST (right).

formal definition is deferred to §5). Our protocol ensures that all valid views are *comparable*. Formally,  $v_1 \subset v_2$  means that  $v_1.changes \subset v_2.changes$ , we say that  $v_2$  is *more recent* than  $v_1$ . Two different views are comparable if one is more recent than the other, otherwise they *conflict*. We assume that the *initial view*, i.e., the set of participants at time 0, is publicly known (Assumption 2).

A valid view  $v$  must be equipped with a *quorum system*: a collection of subsets of  $v.members$ . We choose the quorums to be all subsets of size  $v.q = |v| - \lfloor \frac{|v|-1}{3} \rfloor$ .

► **Assumption 3** (Quorum systems). *In every valid view  $v$ , the number of Byzantine processes is less than or equal to  $\lfloor \frac{|v|-1}{3} \rfloor$  and at least one quorum in  $v$  contains only correct processes.*

Thus, every two quorums of a valid view have a correct process in common and at least one quorum contains only correct processes.<sup>2</sup>

**Sequence of views.** We now build upon the comparability of views to obtain the abstraction of a *sequence of views*, or just *sequence*. A sequence  $seq$  is a set of mutually comparable views. Note that a set with just one view is, trivially, a sequence of views, so is the empty set.

**Reliable multicast.** In addition to the use of signed messages (§2.1), we build our algorithm on top of an elementary (static) reliable Byzantine broadcast protocol. We instantiate this protocol from a standard solution in the literature for a static set of processes, as described e.g., in [9]. The terms “*R-multicast*” and “*R-delivery*” refer to the request to broadcast a message and deliver a message via this protocol to (or from) a static set of processes. For completeness, we provide the pseudocode of the static reliable Byzantine broadcast primitive in [14].

### 3.2 DBRB-JOIN and DBRB-LEAVE Operations

Upon invoking the DBRB-JOIN operation, a process  $p$  first learns the current membership – i.e., the most recent view  $v$  – of the broadcast system through a *View Discovery* protocol (§4.1). The joining operation then consists of four steps (the left part of Figure 2). First,  $p$  disseminates a  $\langle \text{RECONFIG}, \langle +, p \rangle \rangle$  message to members of  $v$ . In the second step, when any correct process  $q$  from  $v$  receives the RECONFIG message,  $q$  proposes to change the system membership to a view  $v'$ , where  $v'$  is an extension of  $v$  including the change  $\langle +, p \rangle$ . To do so,  $q$  disseminates to members of  $v$  a PROPOSE message, containing the details of  $v'$ . Third, any other correct member in  $v$  waits until  $v.q$  matching PROPOSE messages (a quorum

<sup>2</sup> Note that this bound applies both to processes that are active participants, as well as processes leaving the system. This requirement can be relaxed in practice by enforcing a correct process that leaves the system to destroy its private key. Even if the process is later compromised, it will not be able to send any protocol messages. Note that we assume that messages sent while the process was correct cannot be withdrawn or modified.

of  $v$  confirms the new view). Once a process collects the confirmation, it disseminates a  $\langle \text{CONVERGED}, v' \rangle$  message to members of  $v$ . This concludes step three. In the fourth step, each correct process  $q$  in  $v$  waits to gather matching  $\text{CONVERGED}$  messages from a quorum (i.e.,  $v.q$ ) of processes. We say that processes that are members of view  $v$  are trying to *converge* on a new membership. Then,  $q$  triggers an  $R$ -multicast of the  $\langle \text{INSTALL}, v' \rangle$  message to members of  $v \cup v'$ ; recall that the process  $p$  belongs to  $v'$ . Upon  $R$ -delivery of an  $\text{INSTALL}$  message for  $v'$ , any process  $q$  updates its current view to  $v'$ . The  $\text{DBRB-JOIN}$  operation finishes at process  $p$  once this process receives the  $\text{INSTALL}$  message for  $v' \ni p$ . From this instant on,  $p$  is a participant in the system.

The steps executed after a correct process  $p$  invokes the  $\text{DBRB-LEAVE}$  operation are almost identical, except for the fact that  $p$  still executes its “duties” in  $\text{DBRB}$  until  $\text{DBRB-LEAVE}$  returns.<sup>3</sup>

### 3.3 DBRB-BROADCAST Operation

A correct process  $s$  that invokes  $\text{DBRB-BROADCAST}(m)$  first disseminates a  $\text{PREPARE}$  message to every member of the  $s$ ' current view  $v$ . When a correct process  $q$  receives this message,  $q$  sends an  $\text{ACK}$  message to  $s$ , representing a signed statement asserting that  $q$  indeed received  $m$  from  $s$ . Once  $s$  collects a quorum of matching  $\text{ACK}$  messages for  $m$ ,  $s$  constructs a *message certificate*  $\Sigma$  out of the collected signatures  $\rho$ , and disseminates this certificate to every member of  $v$  as part of a  $\text{COMMIT}$  message. When any correct process  $q$  receives a  $\text{COMMIT}$  message with a valid certificate for  $m$  for the first time,  $q$  relays this message to all members of view  $v$ . Moreover,  $q$  sends a  $\text{DELIVER}$  message to the sender of the  $\text{COMMIT}$  message. Once any process  $q$  collects a quorum of matching  $\text{DELIVER}$  messages,  $q$  triggers  $\text{DBRB-DELIVER}(m)$ . The right part of Figure 2 presents the overview of this operation. In Figure 2, we depict process  $s$  collecting enough  $\text{DELIVER}$  messages to deliver  $m$ , assuming that all processes in the system use the same view. The details of how views are changed during an execution of a broadcast operation are given in §4.2.

## 4 DBRB Algorithm

In this section, we describe our  $\text{DBRB}$  algorithm, starting with dynamic membership (§4.1), and continuing with broadcast (§4.2). We also present an illustrative execution of  $\text{DBRB}$  (§4.3).

Algorithm 1 introduces the variables that each process  $p$  maintains, as well as two helper functions to compute the least recent and most recent view of a given sequence, respectively.

### 4.1 Dynamic Membership

Algorithms 2 and 3 contain the pseudocode of the  $\text{DBRB-JOIN}$  and  $\text{DBRB-LEAVE}$  operations. Let us first discuss the join operation.

After a correct process  $p$  invokes the  $\text{DBRB-JOIN}$  operation,  $p$  obtains the most recent view of the system, and it does so through the View Discovery protocol. We describe the View Discovery protocol at the end of this section; for the moment it suffices to say that  $p$  obtains the most recent view  $v$  and updates its local variable  $cv$  to reflect this view. Next, process  $p$  disseminates a  $\langle \text{RECONFIG}, \langle +, p \rangle, cv \rangle$  message to every member of  $cv$  (Algorithm 2)

<sup>3</sup> There is a detail we deliberately omitted from this high-level description and we defer to §4.1: multiple processes may try to join the system concurrently, and thereby multiple  $\text{PROPOSE}$  messages may circulate at the same time. These messages comprise different views, e.g., one could be for a view  $v'$  and another for  $v''$ . These conflicts are unavoidable in asynchronous networks. For this reason,  $\text{PROPOSE}$  messages (and other protocol messages) operate at the granularity of sequences, not individual views. If conflicts occur, sequences support union and ordering, allowing reconciliation of  $v'$  with  $v''$  on a sequence that comprises their union.

notifying members of  $cv$  of its intention to join. The view discovery and the dissemination are repeated until the *joinComplete* event triggers or a quorum of confirmation messages has been collected for some view  $v$  to which RECONFIG message was broadcast (Algorithm 2).

Every correct member  $r$  of the view  $cv$  proposes a new system membership that includes process  $p$ , once  $r$  receives the aforementioned RECONFIG message from process  $p$ . The new proposal is incorporated within a *sequence* of views  $SEQ^v$ ,  $v = cv$ , (containing, initially, just one view) and disseminated to all members of the view  $cv$  via a PROPOSE message (Algorithm 2).

The leaving operation invocation is similar: Process  $p$  disseminates a RECONFIG message with  $\langle -, p \rangle$  as an argument, and process  $r$  proposes a new system membership that *does not* include  $p$ . The main difference with the joining operation is that if  $p$  delivered or is the sender of a message,  $p$  must ensure validity and totality properties of DBRB before disseminating a RECONFIG message (Algorithm 2).

Let us now explain how a new view is installed in the system. The correct process  $r \in cv$  receives PROPOSE messages disseminated by other members of  $cv$ . First,  $r$  checks whether it *accepts*<sup>4</sup> the received proposal (recall that a proposal is a sequence of views). Moreover,  $r$  checks whether the received proposed sequence  $seq$  is well-formed, i.e., whether  $seq$  satisfies the following: (1)  $seq$  is a sequence of views, (2) there is at least one view in  $seq$  that  $r$  is not aware of, and (3) every view in  $seq$  is more recent than  $cv$ .

If all the checks have passed, the process  $r$  uses the received PROPOSE message to update its own proposal. This is done according to two cases:

1. There are conflicts between  $r$ 's and the received proposal (Algorithm 2 to Algorithm 2). In this case,  $r$  creates a new proposal containing  $r$ 's last converged sequence for the view<sup>5</sup> and a new view representing the union of the most recent views of two proposals.
2. There are no conflicts (Algorithm 2). In this case,  $r$  executes the union of its previous and received proposal in order to create a new proposal.

Once  $r$  receives the same proposal from a quorum of processes,  $r$  updates its last converged sequence (Algorithm 2) and disseminates it within a CONVERGED message (Algorithm 2).

When  $r$  receives a CONVERGED message for some sequence of views  $seq'$  and some view  $v$  (usually  $v$  is equal to the current view  $cv$  of process  $r$ , but it could also be a less recent view than  $cv$ ) from a quorum of members of the view  $v$  (Algorithm 2),  $r$  creates and reliably disseminates an INSTALL message that specifies the view that should be replaced (i.e.,  $v$ ), the least recent view of the sequence  $seq'$  denoted by  $\omega$  (Algorithm 2) and the entire sequence  $seq'$  (Algorithm 2). Moreover, we say that  $seq'$  is *converged on* to replace  $v$ . An INSTALL message is disseminated to processes that are members of views  $v$  or  $\omega$  (Algorithm 2). Note that INSTALL messages include a quorum of signed CONVERGED messages which ensures its authenticity (omitted in Algorithms 2 and 3 for brevity).

Once the correct process  $r$  receives the INSTALL message (Algorithm 3),  $r$  enters the installation procedure in order to update its current view of the system. There are four parts to consider:

1. Process  $r$  was a member of a view  $v$  (Algorithm 3): Firstly,  $r$  checks whether  $cv \subset \omega$ , where  $cv$  is the current view of  $r$ . If this is the case,  $r$  stops processing PREPARE, COMMIT

<sup>4</sup> Process  $r$  accepts a sequence of views  $seq$  to replace a view  $v$  if  $seq \in FORMAT^v$  or  $\emptyset \in FORMAT^v$  (Algorithm 2). The following holds at every correct process that is a member of the initial view of the system  $v_0$ :  $\emptyset \in FORMAT^{v_0}$ . Note that  $FORMAT^v$ , for any view  $v$ , is a set of sequences, i.e., a set of sets.

<sup>5</sup> We say that  $seq$  is the last converged sequence for a view  $v$  of a process if the process receives the same proposal to replace the view  $v$  from a quorum of members of  $v$  (variable  $LCSEQ^v$ ).

■ **Algorithm 1** DBRB algorithm: local variables of process  $p$  and helper functions.

---

```

1: variables:
2:    $cv = v_0$  // current view;  $v_0$  is the initial view
3:    $RECV = \emptyset$  // set of pending updates (i.e., join or leave)
4:    $SEQ^v = \emptyset$  // set of proposed sequences to replace  $v$ 
5:    $LCSEQ^v = \emptyset$  // last converged sequence to replace  $v$ 
6:    $FORMAT^v = \emptyset$  // replacement sequence for view  $v$ 
7:    $cer = \perp$  // message certificate for  $m$ 
8:    $v_{cer} = \perp$  // view in which certificate is collected
9:    $\triangleright$  set of messages allowed to be acknowledged; initially, any message could be acknowledged by a
   process
10:   $allowed\_ack = \perp$  //  $\perp$  - any message,  $\top$  - no message
11:   $stored = false$ ;  $stored\_value = \perp$ 
12:   $can\_leave = false$  // process is allowed to leave
13:   $delivered = false$  //  $m$  delivered or not
14:   $\triangleright$  for every process  $q \in \mathcal{U}$  and every valid view  $v$ 
15:   $acks[q, v] = \perp$ ;  $\Sigma[q, v] = \perp$ ;  $deliver[q, v] = \perp$ 
16:   $State = \perp$  // state of the process; consists of ack, conflicting and stored fields

17: function least_recent( $seq$ ) returns  $\omega \in seq : \nexists \omega' \in seq : \omega' \subset \omega$ 
18: function most_recent( $seq$ ) returns  $\omega \in seq : \nexists \omega' \in seq : \omega \subset \omega'$ 

```

---

and RECONFIG messages (Algorithm 3; see §4.2). Therefore, process  $r$  will not send any ACK or DELIVER message for PREPARE or COMMIT messages associated with  $v$  (and views preceding  $v$ ). The same holds for RECONFIG messages. We refer to acknowledged and stored messages by a process as the *state* of the process (represented by the *State* variable). The fact that  $r$  stops processing the aforementioned messages is important because  $r$  needs to convey this information via the STATE-UPDATE message (Algorithm 3) to the members of the new view  $\omega$ . Therefore, a conveyed information is “complete” since a correct process  $r$  will never process any PREPARE, COMMIT or RECONFIG message associated with “stale” views (see §4.2).

2. View  $\omega$  is more recent than  $r$ 's current view  $cv$  (Algorithm 3 to Algorithm 3): Process  $r$  waits for  $v.q$  of STATE-UPDATE messages (Algorithm 3) and processes received states (Algorithm 3). STATE-UPDATE messages carry information about: (1) a message process is allowed to acknowledge (*allowed\_ack* variable), (2) a message stored by a process (*stored\_value* variable), and (3) reconfiguration requests observed by a process (see §4.2). Hence, a STATE-UPDATE message contains at most two PREPARE messages associated with some view and properly signed by  $s$  (corresponds to (1)). Two PREPARE messages are needed if a process observes that  $s$  broadcast two messages and are used to convince other processes not to acknowledge any messages (variable *State.conflicting*; Algorithm 4). Moreover, STATE-UPDATE messages contain at most one COMMIT message associated with some view with a valid message certificate (variable *State.stored*) and properly signed by  $s$  (corresponds to (2)), and a (possibly empty) list of properly signed RECONFIG messages associated with some installed view (corresponds to (3)). Note that processes include only PREPARE, COMMIT and RECONFIG messages associated with some view  $v'' \subseteq v$  in the STATE-UPDATE message they send (incorporated in the *state(v)* function). The reason is that processes receiving these STATE-UPDATE messages may not know whether views  $v'' \supset v$  are indeed “created” by our protocol and not “planted” by faulty processes.
3. Process  $r$  is a member of  $\omega \supset cv$  (Algorithm 3 to Algorithm 3): If this is the case,  $r$  updates its current view (Algorithm 3). Moreover,  $r$  *installs* the (updated) current view  $cv$  if the sequence received in the INSTALL message does not contain other views that are more recent than  $cv$  (Algorithm 3).
4. Process  $r$  is not a member of  $\omega \supset cv$  (Algorithm 3 to Algorithm 3): A leaving process  $r$  executes the View Discovery protocol (Algorithm 3) in order to ensure totality of DBRB (we explain this in details in §4.2). When  $r$  has “fulfilled” its role in ensuring totality of DBRB,  $r$  leaves the system (Algorithm 3).



---

**Algorithm 2** DBRB-JOIN and DBRB-LEAVE implementations at process  $p$ .
 

---

```

19: procedure DBRB-JOIN()
20:   repeat
21:      $cv = \text{view\_discovery}(cv)$ 
22:     disseminate  $\langle \text{RECONFIG}, \langle +, p \rangle, cv \rangle$  to all  $q \in cv.\text{members}$ 
23:     until  $\text{joinComplete}$  is triggered or  $v.q \langle \text{REC-CONFIRM}, v \rangle$  messages collected for some  $v$ 
24:     wait for  $\text{joinComplete}$  to be triggered

25: procedure DBRB-LEAVE()
26:   if  $\text{delivered} \vee p = s$  then wait until  $\text{can\_leave}$ 
27:   repeat in each installed view  $cv$  do // in each subsequent view  $p$  installs
28:     disseminate  $\langle \text{RECONFIG}, \langle -, p \rangle, cv \rangle$  to all  $q \in cv.\text{members}$ 
29:     until  $\text{leaveComplete}$  is triggered or  $v.q \langle \text{REC-CONFIRM}, v \rangle$  messages collected for some  $v$ 
30:     wait for  $\text{leaveComplete}$  to be triggered

31: upon receipt of  $\langle \text{RECONFIG}, \langle c, q \rangle, v \rangle$  from  $q$  //  $c \in \{-, +\}$ 
32:   if  $v = cv \wedge \langle c, q \rangle \notin v \wedge (\text{if } (c = -) \text{ then } \langle +, q \rangle \in v)$  then
33:      $\text{RECV} = \text{RECV} \cup \{\langle c, q \rangle\}$ 
34:     send  $\langle \text{REC-CONFIRM}, cv \rangle$  to  $q$ 
35:   end if

36: upon  $\text{RECV} \neq \emptyset \wedge \text{installed}(cv)$  do
37:   if  $\text{SEQ}^{cv} = \emptyset$  then
38:      $\text{SEQ}^{cv} = \{cv \cup \text{RECV}\}$ 
39:     disseminate  $\langle \text{PROPOSE}, \text{SEQ}^{cv}, cv \rangle$  to all  $q \in cv.\text{members}$ 
40:   end if

41: upon receipt of  $\langle \text{PROPOSE}, seq, v \rangle$  from  $q \in v.\text{members}$  such that  $seq \in \text{FORMAT}^v \vee \emptyset \in \text{FORMAT}^v$ 
42:   if  $\text{valid}(seq)$  then // filter incorrect proposals
43:     if  $\text{conflicting}(seq, \text{SEQ}^v)$  then
44:        $\omega = \text{most\_recent}(seq)$ 
45:        $\omega' = \text{most\_recent}(\text{SEQ}^v)$ 
46:        $\triangleright$  merge the last view from the local and  $q$ 's proposal
47:        $\text{SEQ}^v = \text{LCSEQ}^v \cup \{\omega \cup \omega'\}$ 
48:     else // no conflicts, just merge the proposals
49:        $\text{SEQ}^v = \text{SEQ}^v \cup seq$ 
50:     end if
51:     disseminate  $\langle \text{PROPOSE}, \text{SEQ}^v, v \rangle$  to all  $q' \in v.\text{members}$ 
52:   end if

53: upon receipt of  $\langle \text{PROPOSE}, \text{SEQ}^v, v \rangle$  from  $v.q$  processes in  $v$ 
54:    $\text{LCSEQ}^v = \text{SEQ}^v$ 
55:   disseminate  $\langle \text{CONVERGED}, \text{SEQ}^v, v \rangle$  to all  $q \in v.\text{members}$ 

56: upon receipt of  $\langle \text{CONVERGED}, seq', v \rangle$  from  $v.q$  processes in  $v$ 
57:    $\omega = \text{least\_recent}(seq')$ 
58:    $R\text{-multicast}(\{j : j \in v.\text{members} \vee j \in \omega.\text{members}\}, \langle \text{INSTALL}, \omega, seq', v \rangle)$ 

```

---

**View Discovery.** Views “created” during an execution of DBRB form a sequence (see [14]). The View Discovery subprotocol provides information about the sequence of views incorporated in an execution so far. Since every correct process in the system knows the initial view (Assumption 2) and valid transition between views implies the existence of an INSTALL message with a quorum of properly signed CONVERGED messages, any sequence of views starting from the initial view of the system such that appropriate INSTALL messages “connect” adjacent views can be trusted.

A correct process that has invoked the DBRB-JOIN operation and has not left the system executes the View Discovery subprotocol constantly. Once a correct process starts trusting a sequence of views, it disseminates that information to all processes in the universe. A correct process executing the View Discovery subprotocol learns which sequences of views are trusted by other processes. Once the process observes a sequence of views allegedly trusted by a process, it can check whether the sequence is properly formed (as explained above) and if that is the case, the process can start trusting the sequence and views incorporated in it (captured by the `view_discovery` function for the joining and leaving process; Algorithms 2 and 3).



■ **Algorithm 3** DBRB algorithm: installing a view at process  $p$ .

---

```

59: upon  $R$ -delivery( $\{j : j \in v.members \vee j \in \omega.members\}, \langle \text{INSTALL}, \omega, seq, v \rangle$ ) do
60:    $FORMAT^\omega = FORMAT^\omega \cup \{seq \setminus \{\omega\}\}$ 
61:   if  $p \in v.members$  then //  $p$  was a member of  $v$ 
62:     if  $cv \subset \omega$  then stop processing PREPARE, COMMIT and RECONFIG messages
63:      $R$ -multicast( $\{j : j \in v.members \vee j \in \omega.members\}, \langle \text{STATE-UPDATE}, state(v), RECV \rangle$ )
64:   end if
65:   if  $cv \subset \omega$  then //  $\omega$  is more recent than  $p$ 's current view
66:     wait for  $\langle \text{STATE-UPDATE}, *, * \rangle$  messages from  $v.q$  processes in  $v$  // from the reliable broadcast
67:      $req = \{\text{reconfiguration requests from STATE-UPDATE messages}\}$ 
68:      $RECV = RECV \cup (req \setminus \omega.changes)$ 
69:      $states = \{\text{states from STATE-UPDATE messages}\}$ 
70:      $installed(\omega) = \text{false}$ 
71:     invoke  $state\text{-transfer}(states)$  // Algorithm 4
72:     if  $p \in \omega.members$  then //  $p$  is in  $\omega$ 
73:        $cv = \omega$ 
74:       if  $p \notin v.members$  then trigger  $joinComplete$  // can return from DBRB-JOIN
75:       if  $\exists \omega' \in seq : cv \subset \omega'$  then
76:          $seq' = \{\omega' \in seq : cv \subset \omega'\}$ 
77:         if  $SEQ^{cv} = \emptyset \wedge \forall \omega \in seq' : cv \subset \omega$  then
78:            $SEQ^{cv} = seq'$ 
79:           disseminate  $\langle \text{PROPOSE}, SEQ^{cv}, cv \rangle$  to all  $q \in cv.members$ 
80:         end if
81:       else
82:          $installed(cv) = \text{true}$ 
83:         resume processing PREPARE, COMMIT and RECONFIG messages
84:         invoke  $new\text{-view}()$  // Algorithm 4
85:       end if
86:     else //  $p$  is leaving the system
87:       if  $stored$  then
88:         while  $\neg can\_leave$  do
89:            $cv = \text{view\_discovery}(cv)$ 
90:           disseminate  $\langle \text{COMMIT}, m, cer, v_{cer}, cv \rangle$  to all  $q \in cv.members$ 
91:         end while
92:       end if
93:       trigger  $leaveComplete$  // can return from DBRB-LEAVE
94:     end if
95:   end if

```

---

The View Discovery protocol addresses two main difficulties: (1) it enables processes joining and leaving the system to learn about the current membership of the system, (2) it is crucial to ensure the consistency, validity and totality properties of DBRB since it supplies information about views “instantiated” by the protocol and associated quorum systems. We formally discuss the View Discovery protocol in the full version of the paper [14].

## 4.2 Broadcast

In order to broadcast some message  $m$ , processes in DBRB use the following types of messages: PREPARE: When a correct process  $s$  invokes a DBRB-BROADCAST( $m$ ) operation, the algorithm creates a  $m_{prepare} = \langle \text{PREPARE}, m, cv_s \rangle$  message, where  $cv_s$  is the current view of the system of process  $s$ . Message  $m_{prepare}$  is sent to every process that is a member of  $cv_s$  (Algorithm 4). Process  $s$  disseminates the PREPARE message if  $cv_s$  is installed by  $s$ ; otherwise,  $s$  does not disseminate the message to members of  $cv_s$  (Algorithm 4), but rather waits to install some view and then disseminates the PREPARE message (Algorithm 4).

ACK: When a correct process  $q$  receives  $m_{prepare}$  message,  $q$  firstly checks whether view specified in  $m_{prepare}$  is equal to the current view of  $q$  (Algorithm 4). If that is the case,  $q$  checks whether it is allowed to send an ACK message for  $m$  (see Consistency paragraph in §5; Algorithm 4) and if it is,  $q$  sends  $m_{ack} = \langle \text{ACK}, m, \sigma, cv_q \rangle$  message to process  $s$  (i.e., the sender of  $m_{prepare}$ ), where  $\sigma$  represents the signed statement that  $s$  sent  $m$  to  $q$  (Algorithm 4).

■ **Algorithm 4** DBRB-BROADCAST( $m$ ) and DBRB-DELIVER( $m$ ) implementations at process  $p$ .

---

```

96: procedure state-transfer(states)
97:   if (allowed_ack =  $\perp$   $\vee$  allowed_ack =  $m$ )  $\wedge$   $m$  is the only acknowledged message among states
then
98:     allowed_ack =  $m$ ; update_if_bot(State.ack, prepare_msg) // updated only if it is  $\perp$ 
99:   else if there exist at least two different messages acknowledged among states then
100:      $\triangleright$   $p$  and  $p'$  are different PREPARE messages
101:     allowed_ack =  $\top$ ; update_if_bot(State.conflicting,  $p, p'$ ); State.ack =  $\perp$ 
102:   else if there exists a state among states such that it provides two different broadcast messages
then
103:     allowed_ack =  $\top$ ; update_if_bot(State.conflicting,  $p, p'$ ); State.ack =  $\perp$ 
104:   end if
105:   if  $\neg$ stored  $\wedge$  there exists a stored message  $m$  with a valid message certificate among states then
106:     stored = true; stored_value = ( $m, cer, v_{cer}$ ) // cer is the message certificate collected in view
107:     update_if_bot(State.stored, commit_msg) // updated only if it is  $\perp$ 
108:   end if
109: procedure new-view()
110:   if  $p = s \wedge cer = \perp$  then disseminate  $\langle$ PREPARE,  $m, cv$  $\rangle$  to all  $q \in cv.members$ 
111:   if  $p = s \wedge cer \neq \perp \wedge \neg can\_leave$  then disseminate  $\langle$ COMMIT,  $m, cer, v_{cer}, cv$  $\rangle$  to all  $q \in$ 
112:     cv.members
113:   if  $p \neq s \wedge stored \wedge \neg can\_leave$  then disseminate  $\langle$ COMMIT,  $m, cer, v_{cer}, cv$  $\rangle$  to all  $q \in cv.members$ 
113: procedure DBRB-BROADCAST( $m$ )
114:   if installed(cv) then disseminate  $\langle$ PREPARE,  $m, cv$  $\rangle$  to all  $q \in cv.members$ 
115: upon receipt of  $\langle$ PREPARE,  $m, v$  $\rangle$  from  $s \in v.members$  such that  $v = cv$ 
116:   if allowed_ack =  $m \vee allowed\_ack = \perp$  then
117:     allowed_ack =  $m$ ; update_if_bot(State.ack,  $\langle$ PREPARE,  $m, v$  $\rangle$ ) // updated only if it is  $\perp$ 
118:      $\sigma = sign(m, cv)$ ; send  $\langle$ ACK,  $m, \sigma, cv$  $\rangle$  to  $s$ 
119:   end if
120: upon receipt of  $\langle$ ACK,  $m, \sigma, v$  $\rangle$  from  $q \in v.members$  // only process  $s$ 
121:   if acks[ $q, v$ ] =  $\perp \wedge verify_{sig}(q, m, v, \sigma)$  then acks[ $q, v$ ] =  $m$ ;  $\Sigma[q, v] = \sigma$ 
122: upon exists  $m \neq \perp$  and  $v$  such that  $|\{q \in v.members \mid acks[q, v] = m\}| \geq v.q \wedge cer = \perp$  do
123:   cer =  $\{\Sigma[q, v] : acks[q, v] = m\}$ ;  $v_{cer} = v$ 
124:   if installed(cv) then disseminate  $\langle$ COMMIT,  $m, cer, v_{cer}, cv$  $\rangle$  to all  $q' \in cv.members$ 
125: upon receipt of  $\langle$ COMMIT,  $m, cer, v_{cer}, v$  $\rangle$  from  $q$  such that  $v = cv$ 
126:   if verify_certificate(cer,  $v_{cer}, m$ ) then
127:     if  $\neg stored$  then
128:       stored = true; stored_value = ( $m, cer, v_{cer}$ )
129:       update_if_bot(State.stored,  $\langle$ COMMIT,  $m, cer, v_{cer}, v$  $\rangle$ ) // updated only if it is  $\perp$ 
130:       disseminate  $\langle$ COMMIT,  $m, cer, v_{cer}, cv$  $\rangle$  to all  $q' \in cv.members$ 
131:     end if
132:     send  $\langle$ DELIVER,  $m, cv$  $\rangle$  to  $q$ 
133:   end if
134: upon receipt of  $\langle$ DELIVER,  $m, v$  $\rangle$  from  $q \in v.members$ 
135:   if deliver[ $q, v$ ] =  $\perp$  then deliver[ $q, v$ ] =  $\top$ 
136: upon exists  $v$  such that  $|\{q \in v.members \mid deliver[q, v] = \top\}| \geq v.q$  for the first time do
137:   delivered = true
138:   invoke DBRB-DELIVER( $m$ )
139:   can_leave = true // If  $p = s$ , DBRB-BROADCAST is completed

```

---

When some process  $q$  sends an ACK message for  $m$  ( $m$  is a second argument of the message), we say that  $q$  *acknowledges*  $m$ . Moreover, if an ACK message is associated with some view  $v$ , we say that  $q$  acknowledges  $m$  in a view  $v$ .

COMMIT: When process  $s$  receives a quorum of appropriate ACK messages associated with the same view  $v$  for  $m$  (Algorithm 4),  $s$  collects received signed statements into a *message certificate*. Process  $s$  then creates  $m_{commit} = \langle$ COMMIT,  $m, cer, v_{cer}, cv_s$  $\rangle$  message and disse-

inates  $m_{commit}$  to every process that is a member of  $cv_s$  (Algorithm 4). Note that  $cv_s$  may be different from  $v$  (we account for this in the rest of the section). Moreover,  $s$  disseminates the COMMIT message (Algorithm 4) if  $cv_s$  is installed by  $s$ ; otherwise,  $s$  does not disseminate the message to members of  $cv_s$ , but rather waits to install some view and then disseminates the COMMIT message (Algorithm 4).

**DELIVER:** When a correct process  $q$  receives  $m_{commit}$  message, it firstly checks whether view specified in  $m_{commit}$  is equal to the current view of  $q$  (Algorithm 4). If that is the case and the message certificate is valid (Algorithm 4),  $q$  “stores”  $m$  (Algorithm 4) and sends  $m_{deliver} = \langle \text{DELIVER}, m, cv_q \rangle$  to process  $s$  as an approval that  $s$  can deliver  $m$  (Algorithm 4). When a process  $q$  executes Algorithm 4 or Algorithm 4 for a message  $m$ , we say that  $q$  stores  $m$ . Observe that  $q$  also disseminates  $m_{commit}$  in order to deliver  $m$  itself (Algorithm 4).

Lastly, once a correct process receives a quorum of appropriate DELIVER messages associated with the same view  $v$  for  $m$  (Algorithm 4), it delivers  $m$  (Algorithm 4).

Every PREPARE, ACK, COMMIT and DELIVER message is associated with one specific view. We can divide the broadcasting of message  $m$  by the correct sender  $s$  into two phases:

- **Certificate collection phase:** This phase includes a dissemination of an appropriate PREPARE message and a wait for a quorum of ACK messages by process  $s$ . Note that PREPARE and ACK messages are associated with the same view  $v$ . We say that certificate collection phase is *executed* in view  $v$ . Moreover, if  $s$  indeed receives a quorum of ACK messages associated with  $v$ , we say that certificate collection phase is *successfully executed* in  $v$ . In that case, sometimes we say that  $s$  collects a message certificate in  $v$ .
- **Storing phase:** In this phase, each correct process  $p$  (including  $s$ ) disseminates a COMMIT message (containing a valid message certificate collected in the previous phase), and waits for a quorum of DELIVER messages. Note that COMMIT and DELIVER messages are associated with the same view  $v$ . We say that storing phase is *executed* in view  $v$ . Moreover, if  $p$  indeed receives a quorum of DELIVER messages associated with  $v$ , we say that storing phase is *successfully executed* in  $v$ .

Observe that the certificate collection phase can be successfully executed in some view  $v$ , whereas the storing phase can be executed in some view  $v' \supset v$ . This is the reason why we include  $v_{cer}$  argument in a COMMIT message, representing the view in which a message certificate is collected. Lastly, in order to ensure validity and totality, processes must disseminate PREPARE and COMMIT messages in new views they install until they collect enough ACK and DELIVER messages, respectively. This mechanism is captured in the *new-view* procedure (Algorithm 4) that is invoked when a view is installed (Algorithm 3).

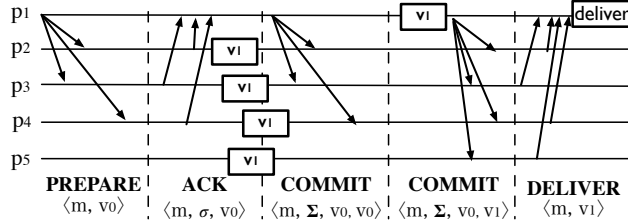
### 4.3 Illustration

Consider four participants at time  $t = 0$ . Process  $p_1$  broadcasts a message  $m$ . Hence,  $p_1$  sends to processes  $p_1, p_2, p_3, p_4$  a  $m_{prepare} = \langle \text{PREPARE}, m, v_0 \rangle$  message, where  $v_0 = \{ \langle +, p_1 \rangle, \langle +, p_2 \rangle, \langle +, p_3 \rangle, \langle +, p_4 \rangle \}$ . Since all processes consider  $v_0$  as their current view of the system at time of receiving of  $m_{prepare}$  message, they send to  $p_1$  an appropriate ACK message and  $p_1$  collects a quorum (with respect to  $v_0$ ) of ACK messages for  $m$ .

However, process  $p_5$  invokes a DBRB-JOIN operation and processes  $p_2, p_3, p_4, p_5$  set  $v_1 = \{ \langle +, p_1 \rangle, \langle +, p_2 \rangle, \langle +, p_3 \rangle, \langle +, p_4 \rangle, \langle +, p_5 \rangle \}$  as their current view of the system. Process  $p_1$  still considers  $v_0$  as its current view of the system and disseminates a  $m_{commit} = \langle \text{COMMIT}, m, cer, v_{cer} = v_0, v_0 \rangle$  message. Processes  $p_2, p_3$  and  $p_4$  do not store  $m$ , since  $v_0$  (specified in  $m_{commit}$  message) is not their current view. Observe that  $p_1$  stores  $m$  since  $v_0$  is still the current view of the system from  $p_1$ 's perspective.

Once process  $p_1$  assigns  $v_1$  as its current view of the system, it disseminates  $m_{commit} = \langle \text{COMMIT}, m, cer, v_{cer} = v_0, v_1 \rangle$  message to processes that are members of  $v_1$  and they all store  $m$  and relay  $m_{commit}$  message to all processes that are members of  $v_1$ . Hence, they

all deliver message  $m$  once they collect a quorum (with respect to  $v_1$ ) of matching DELIVER messages. In this execution  $p_1$  has successfully executed a certificate collection phase in  $v_0$  and then reused the message certificate to relay an appropriate COMMIT message to processes that are members of  $v_1$  (since the system has reconfigured to  $v_1$ ). Note that Figure 3 depicts the described execution. For presentation simplicity, Figure 3 just shows the COMMIT and DELIVER messages that allow process  $p_1$  to deliver  $m$ .



■ **Figure 3** Example of a broadcast operation in DBRB algorithm, considering a dynamic membership.

## 5 DBRB Algorithm Correctness

We now give an intuition of why our DBRB algorithm is correct; we give formal arguments in the full version of the paper [14].

We first define the notions of *valid* and *installed* views. A view  $v$  is *valid* if: (1)  $v$  is the initial view of the system, or (2) a sequence  $seq = v \rightarrow \dots$  is converged on to replace some valid view  $v'$ . A valid view  $v$  is *installed* if a correct process  $p \in v$  processed PREPARE, COMMIT and RECONFIG messages associated with  $v$  during an execution. By default, the initial view of the system is installed. Lastly, our implementation ensures that installed views form a sequence of views.

**Liveness.** DBRB-JOIN and DBRB-LEAVE operations complete because any change “noticed” by a quorum of processes is eventually processed. Intuitively, a sequence can be converged on if a quorum of processes propose that sequence. Moreover, noticed changes are transferred to new valid views. DBRB-BROADCAST operation completes since a correct sender eventually collects a quorum of DELIVER messages associated with an installed view (see the next paragraph).

**Validity.** Recall that we assume a finite number of reconfiguration requests in any execution of DBRB (Assumption 1), which means that there exists a view  $v_{final}$  from which the system will not be reconfigured. In order to prove validity, it suffices to show that every correct member of  $v_{final}$  delivers a broadcast message.

A correct process  $s$  that broadcasts a message  $m$  executes a certificate collection phase in some installed view  $v$  (the current view of  $s$ ). Even if  $s$  does not successfully execute a certificate collection phase in views that precede  $v_{final}$  in the sequence of installed views,  $s$  successfully executes a certificate collection phase in  $v_{final}$ . Note that process  $s$  does not leave the system before it collects enough DELIVER messages (ensured by the check at Algorithm 2 and the assignment at Algorithm 4).

Moreover, a correct process  $p$  that stored a message  $m$  eventually collects a quorum of DELIVER messages associated with some installed view  $v$ . As in the argument above, even if  $p$  does not collect a quorum of DELIVER messages associated with views that precede  $v_{final}$

in the sequence of installed views,  $p$  does that in  $v_{final}$ . Observe that if at least a quorum of processes that are members of some installed view  $v$  store a message  $m$ , then every correct process  $p \in v'$ , where view  $v' \supseteq v$  is installed, stores  $m$ . Let us give the intuition behind this claim. Suppose that view  $v'$  directly succeeds view  $v$  in the sequence of views installed in the system. Process  $p \in v'$  waits for states from at least a quorum of processes that were members of  $v$  (Algorithm 3) before it updates its current view to  $v'$ . Hence,  $p$  receives from at least one process that  $m$  is stored and then  $p$  stores  $m$  (Algorithm 4). The same holds for the correct members of  $v$ .

It now suffices to show that  $s$  collects a quorum (with respect to some installed view) of confirmations that  $m$  is stored, i.e., DELIVER messages. Even if the correct sender does not collect a quorum of DELIVER messages in views that precede  $v_{final}$ , it collects the quorum when disseminating the COMMIT message to members of  $v_{final}$ . Suppose now that the sender collects the aforementioned quorum of DELIVER messages in some installed view  $v$ . If  $v \neq v_{final}$ , every correct member of  $v_{final}$  stores and delivers  $m$  (because of the previous argument). If  $v = v_{final}$ , the reliable communication and the fact that the system can not be further reconfigured guarantee that every correct member of  $v_{final}$  stores and, thus, delivers  $m$ .

**Totality.** The intuition here is similar to that behind ensuring validity. Consider a correct process  $p$  that delivers a message  $m$ :  $p$  successfully executed a storing phase in some installed view  $v$ . This means that every member of an installed view  $v' \supseteq v$  stores  $m$ . Consider a correct participant  $q$  that expressed its will to leave after process  $p$  had delivered  $m$ . This implies that  $q \in v''$ , where  $v'' \supseteq v$  is an installed view, which means that process  $q$  eventually stores  $m$ . As in the previous paragraph, we conclude that process  $q$  eventually collects enough DELIVER messages associated with some installed view and delivers  $m$ .

**Consistency.** A correct process delivers a message only if there exists a message certificate associated with the message (the check at Algorithm 4). Hence, the malicious sender  $s$  must collect message certificates for two different messages in order for the consistency to be violated.

Suppose that process  $s$  has successfully executed a certificate collection phase in some installed view  $v$  for a message  $m$ . Because of the quorum intersection and the verification at Algorithm 4, it is impossible for  $s$  to collect a valid message certificate in  $v$  for some message  $m' \neq m$ . Consider now an installed view  $v'$  that directly succeeds view  $v$  in the sequence of installed views. Since  $s$  collected a message certificate for  $m$  in  $v$ , every correct process  $p \in v'$  receives from at least one process from the view  $v$  that it is allowed to acknowledge only message  $m$  (Algorithm 4). It is easy to see that this holds for every installed view  $v'' \supset v'$ . Therefore, if  $s$  also collects a message certificate for some message  $m'$ , then  $m' = m$  and the consistency holds.

**No duplication.** Trivially follows from Algorithm 4.

**Integrity.** Consider a correct process  $q$  that delivers a message  $m$ . There is a message certificate for  $m$  collected in some installed view  $v$  by  $s$ . A message certificate for  $m$  is collected since a quorum of processes in  $v$  have sent an appropriate ACK message for  $m$ . A correct process sends an ACK message only when it receives an appropriate PREPARE message. Consequently, message  $m$  was broadcast by  $s$ .

## 6 Related Work & Conclusions

**DBRB vs. Static Byzantine Reliable Broadcast.** Our DBRB abstraction generalizes *static* BRB (Byzantine reliable broadcast [8, 22]). Assuming that no process joins or leaves the system, the two abstractions coincide. In a dynamic setting, the validity property of DBRB stipulates that only processes that do not leave the system deliver the appropriate messages. Moreover, the totality property guarantees that only processes that have not expressed their will to leave deliver the message. We prove that stronger variants of these properties are impossible in our model.

**Passive and Active Reconfiguration.** Some reconfigurable systems [6, 3, 4] assume that processes join and leave the system under a specific *churn* model. Intuitively, the consistency properties of the implemented service, e.g., an atomic storage, are ensured assuming that the system does not evolve too quickly and there is always a certain fraction of correct members in the system. In DBRB, we model this through the quorum system assumption on valid views (Assumption 3). Our system model also assumes that booting finishes by time 0 (Assumption 2), thus avoiding the problem of unbounded booting times which could be problematic in asynchronous network [27].

*Active* reconfiguration allows the processes to explicitly propose configuration updates, e.g., sets of new process members. In *DynaStore* [1], reconfigurable dynamic atomic storage is implemented in an asynchronous environment (i.e., without relying on consensus). *Dynastore* implicitly generates a graph of views which provides a way of identifying a sequence of views in which clients need to execute their r/w operations. *SpSn* [12] proposes to capture this order via the *speculating snapshot* algorithm (*SpSn*). *SmartMerge* [17] implements a reconfigurable storage in which not only system membership but also its quorum system can be reconfigured, assuming that a *static* external lattice agreement is available. In [20], it was shown that *reconfigurable lattice agreement* can get rid of this assumption and still implement a large variety of reconfigurable objects. The approach was then extended to the Byzantine fault model [21]. *FreeStore* [2] introduced *view generator*, an abstraction that captures the agreement demands of reconfiguration protocols. Our work is highly inspired by *FreeStore*, which algorithmic and theoretical approach we adapt to an arbitrary failure model.

All reconfigurable solutions discussed above were applied exclusively to shared-memory emulations. Moreover, most of them assumed the crash fault model. In contrast, in this paper, we address the problem of dynamic *reliable broadcast*, assuming an arbitrary (Byzantine) failure model. Also, we do not distinguish between clients and replicas, and assume that every process can only suggest itself as a candidate to join or leave the system. Unlike the concurrent work by Kumar and Welch on Byzantine-tolerant registers [19], our solution can tolerate unbounded number of Byzantine failures, as long as basic quorum assumptions on valid views are maintained.

**Broadcast Applications.** Reliable broadcast is one of the most pervasive primitives in distributed applications [25]. For instance, broadcast can be used for maintaining caches in cloud services [13], or in a publish-subscribe network [11]. Even more interestingly, Byzantine fault-tolerant reliable broadcast (e.g., dynamic solution such as our DBRB, as well as static solutions [8, 16, 23]) are sufficiently strong for implementing decentralized online payments, i.e., cryptocurrencies [15].



**Summary.** This paper presents the specification of DBRB (dynamic Byzantine reliable broadcast), as well as an asynchronous algorithm implementing this primitive. DBRB generalizes traditional Byzantine reliable broadcast, which operates in static environments, to work in a dynamic network. To the best of our knowledge, we are the first to investigate an arbitrary failure model in implementing dynamic broadcast systems. The main merit of our approach is that we did not rely on a consensus building blocks, i.e., DBRB can be implemented completely asynchronously.

---


## References

- 1 Marcos K Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic atomic storage without consensus. *Journal of the ACM (JACM)*, 58(2):7, 2011.
- 2 Eduardo Alchieri, Alysson Bessani, Fabíola Greve, and Joni Fraga. Efficient and modular consensus-free reconfiguration for fault-tolerant storage. *arXiv preprint arXiv:1607.05344*, 2016.
- 3 Hagit Attiya, Hyun Chul Chung, Faith Ellen, Saptarni Kumar, and Jennifer L. Welch. Emulating a shared register in a system that never stops changing. *IEEE Trans. Parallel Distrib. Syst.*, 30(3):544–559, 2019.
- 4 Hagit Attiya, Sweta Kumari, Archit Somani, and Jennifer L. Welch. Store-collect in the presence of continuous churn with application to snapshots and lattice agreement. *CoRR*, abs/2003.07787, 2020. URL: <https://arxiv.org/abs/2003.07787>.
- 5 Roberto Baldoni, Silvia Bonomi, Anne-Marie Kermarrec, and Michel Raynal. Implementing a register in a dynamic distributed system. In *DISC*, pages 639–647. IEEE, 2009.
- 6 Roberto Baldoni, Silvia Bonomi, Anne-Marie Kermarrec, and Michel Raynal. Implementing a register in a dynamic distributed system. In *ICDCS*, pages 639–647, 2009.
- 7 Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.
- 8 Gabriel Bracha and Sam Toueg. Asynchronous Consensus and Broadcast Protocols. *JACM*, 32(4), 1985.
- 9 Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.
- 10 Gregory V Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys (CSUR)*, 33(4):427–469, 2001.
- 11 P. Th. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec. Lightweight probabilistic broadcast. *ACM Trans. Comput. Syst.*, 21(4):341–374, November 2003. doi:10.1145/945506.945507.
- 12 Eli Gafni and Dahlia Malkhi. Elastic configuration maintenance via a parsimonious speculating snapshot solution. In *DISC*, pages 140–153. Springer, 2015.
- 13 Haoyan Geng and Robbert Van Renesse. Sprinkler - Reliable broadcast for geographically dispersed datacenters. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8275 LNCS:247–266, 2013. doi:10.1007/978-3-642-45065-5\_13.
- 14 Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, and Andrei Tonkikh. Dynamic Byzantine Reliable Broadcast [Technical Report]. *arXiv preprint arXiv:2001.06271*, 2020. URL: <https://arxiv.org/abs/2001.06271>.
- 15 Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovič, and Dragos-Adrian Seredinschi. The consensus number of a cryptocurrency. In *PODC*, pages 307–316, 2019.
- 16 Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. Scalable byzantine reliable broadcast. In *DISC*, pages 22:1–22:16, 2019.
- 17 Leander Jehl, Roman Vitenberg, and Hein Meling. Smartmerge: A new approach to reconfiguration for atomic storage. In *International Symposium on Distributed Computing*, pages 154–169. Springer, 2015.



- 18 Anne-Marie Kermarrec and Maarten Van Steen. Gossiping in distributed systems. *ACM SIGOPS operating systems review*, 41(5):2–7, 2007.
- 19 Saptarni Kumar and Jennifer L. Welch. Byzantine-tolerant register in a system with continuous churn. *CoRR*, abs/1910.06716, 2019. [arXiv:1910.06716](https://arxiv.org/abs/1910.06716).
- 20 Petr Kuznetsov, Thibault Rieutord, and Sara Tucci-Piergiovanni. Reconfigurable lattice agreement and applications. In *OPODIS*, 2019.
- 21 Petr Kuznetsov and Andrei Tonkikh. Asynchronous reconfiguration with byzantine failures. In *DISC*, pages 27:1–27:17, 2020.
- 22 Dahlia Malkhi, Michael Merritt, and Ohad Rodeh. Secure Reliable Multicast Protocols in a WAN. In *ICDCS*, 1997.
- 23 Dahlia Malkhi and Michael Reiter. A high-throughput secure reliable multicast protocol. *Journal of Computer Security*, 5(2):113–127, 1997.
- 24 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Whitepaper*, 2008.
- 25 Fernando Pedone and André Schiper. Handling message semantics with generic broadcast protocols. *Distributed Computing*, 15(2):97–107, 2002.
- 26 Alexander Spiegelman and Idit Keidar. On liveness of dynamic storage. In *SIROCCO*, pages 356–376. Springer, 2017.
- 27 Josef Widder and Ulrich Schmid. Booting clock synchronization in partially synchronous systems with hybrid process and link failures. *Distributed Computing*, 20(2):115–140, 2007.


# Broadcasting with Mobile Agents in Dynamic Networks

Shantanu Das 

Aix-Marseille Université, CNRS, Université de Toulon, LIS, Marseille, France  
shantanu.das@lis-lab.fr

Nikos Giachoudis

DCSBI, University of Thessaly, Lamia, Greece  
ngiachou@gmail.com

Flaminia L. Luccio 

DAIS, Università Ca' Foscari Venezia, Italy  
luccio@unive.it

Euripides Markou

DCSBI, University of Thessaly, Lamia, Greece  
emarkou@dib.uth.gr

---

## Abstract

---

We study the standard communication problem of broadcast for mobile agents moving in a network. The agents move autonomously in the network and can communicate with other agents only when they meet at a node. In this model, broadcast is a communication primitive for information transfer from one agent, the source, to all other agents. Previous studies of this problem were restricted to static networks while, in this paper, we consider the problem in dynamic networks modelled as an evolving graph. The dynamicity of the graph is unknown to the agents; in each round an adversary selects which edges of the graph are available, and an agent can choose to traverse one of the available edges adjacent to its current location. The only restriction on the adversary is that the subgraph of available edges in each round must span all nodes; in other words the evolving graph is constantly connected. The agents have global visibility allowing them to see the location of other agents in the graph and move accordingly. Depending on the topology of the underlying graph, we determine how many agents are necessary and sufficient to solve the broadcast problem in dynamic networks. While two agents plus the source are sufficient for ring networks, much larger teams of agents are necessary for denser graphs such as grid graphs and hypercubes, and finally for complete graphs of  $n$  nodes at least  $n - 2$  agents plus the source are necessary and sufficient. We show lower bounds on the number of agents and provide some algorithms for solving broadcast using the minimum number of agents, for various topologies.

**2012 ACM Subject Classification** Networks → Network algorithms; Theory of computation → Distributed algorithms; Theory of computation → Graph algorithms analysis

**Keywords and phrases** Distributed Algorithm, Dynamic Networks, Mobile Agents, Broadcast, Constantly Connected, Global visibility

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2020.24

**Funding** Partially supported by DAIS - Ca' Foscari University of Venice (IRIDE program). Most of this work was done when Shantanu Das and Euripides Markou visited Ca' Foscari University.

## 1 Introduction

We are interested in communication problems for mobile agents moving in a network. The classical problems of broadcast or convergecast deal with the dissemination of information in the network. In the case of message passing networks, broadcast is achieved by spreading the information from the source node to all other nodes. For a system of mobile agents,



© Shantanu Das, Nikos Giachoudis, Flaminia L. Luccio, and Euripides Markou; licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 24; pp. 24:1–24:16

Leibniz International Proceedings in Informatics



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

the equivalent problem is the propagation of information from one source agent to all other agents in the system. Such problems are relevant for teams of mobile sensor robots sent on data collection missions. We assume that the agents autonomously move along the edges of a graph that represents the network; when two agents are at the same node, they can communicate and share information. We would like to stress here that the agents are not allowed to use any means of communicating at a distance (e.g., due to security reasons). The information to be broadcast can be transferred only when agents meet physically.

The problem of broadcasting has been originally investigated in message passing multi-hop radio networks (see e.g. [33, 19]). Previous studies on broadcast and other communication problems have focussed on the efficiency of performing the task, either in terms of time taken [6], or in terms of energy expended [9]. A slightly different line of research considers the problem of broadcast in the presence of faults and the objective is to tolerate as many faults as possible. The faults can be missing links or nodes [20] in the network or loss of messages [21], in case of message passing networks. Recently there has been a lot of interest in so called *dynamic networks* which model both faults and changes in network topology in a uniform manner by considering that the network may change in each round during the execution of the algorithm. The *evolving graph* model [17] represents a dynamic network by a sequence of graphs  $\mathcal{G} = G_1, G_2, \dots$  based on the same set of nodes  $V$  but the set of edges changes in each round  $i$ , i.e., each graph  $G_i = (V, E_i)$  is a spanning subgraph of the underlying graph  $G = (V, \cup E_i)$ , which is called the footprint of the dynamic network. For solving most problems, some assumptions about the connectivity of the dynamic network need to be made. In this paper, we consider the model of *constantly connected* dynamic networks where in round  $i$ , the graph  $G_i$  is assumed to be connected. No other assumptions are made about the network. This means that in some cases, the graphs  $G_i$  and  $G_{i+1}$  in two consecutive rounds may differ completely in the set of available edges. To show correctness of our algorithms, we will assume that an adversary having knowledge of the algorithm, chooses the graph  $G_i$  in each round (respecting the connectivity constraint). This assumption is based on a worst case scenario and similar to the model of  $T$ -interval connected networks studied previously [26, 27], where the network is assumed to contain a stable spanning tree for a continuous period of  $T$  rounds (with  $T = 1$  in our case).

When the underlying graph  $G$  is sparse, there are fewer edges so the adversary has less choice about possible changes in the network, while if  $G$  is a dense graph, the adversary has more choice about which subset of edges to make available, and it can drastically change the network which makes it difficult to design algorithms for these cases. We will consider different network topologies, including sparse topologies, e.g., when  $G$  is a ring or a cactus, as well as denser topologies, e.g., when  $G$  is a grid, a hypercube or a complete graph. For each topology we will design algorithms to solve the broadcast problem under the assumption that agents have global visibility allowing them to see the entire graph and the location of other agents in any round of the algorithm. We denote by  $k$ , the number of agents, other than the source that participate in the algorithm. We will show that the problem becomes easier when there are more agents since at least one of them may be able to reach the source agents and thus make progress in the propagation of the message. In fact we show tight lower bounds on the value of  $k$  necessary to solve the broadcast problem in various topologies. We formally define the problem as follows:

► **Problem 1** (The broadcast problem). *Given a constantly connected dynamic network  $\mathcal{G}$  based on an underlying graph  $G$  consisting of  $n \geq 2$  nodes, a source agent that has a message  $\mathcal{M}$  and  $k \geq 1$  other agents that are initially located at distinct nodes of the network, the goal is to broadcast this message  $\mathcal{M}$  to all the agents.*

## Related Work

Dynamic networks using the evolving graph model or the equivalent notion of time-varying graph model have been studied in [5, 23, 28, 30]. Earlier studies have been devoted to message passing networks in a dynamic setting, often assuming that the future changes to the network are known a priori to the algorithm designer, or, that the dynamicity follows a predefined pattern of repetition as, e.g., in [4]. In message passing dynamic networks, the problem of information dissemination has been studied in [2, 4, 7, 27, 32]. The investigation of mobile agents on dynamic networks started only recently. When agents know the dynamicity of the graph, the problem of exploring a graph in the fastest possible way has been studied in several papers, e.g., [16, 31]. In the case of constantly connected dynamic graphs, the exploration problem has been solved by Ilcinkas et al. [26, 24] in  $\mathcal{O}(n)$  rounds for rings and in  $2^{\mathcal{O}(\sqrt{n})}n$  rounds for cactus graphs.

The scenario when agents do not know the dynamicity of the graph as in this paper, has been mostly studied under restrictive assumptions about the dynamicity, such as periodic [18, 25] or recurrent [26] graphs; while another line of research has looked at probabilistic dynamicity [34]. In the adversarial model (also called the *unknown adversary* model), the adversary chooses the dynamicity of the network, and the agents have no prior knowledge about it. This scenario is the most challenging for designing algorithms and most prior work under this model has been for very simple topologies, namely rings and tori. In this setting, the problem of exploration with termination has been studied for constantly connected rings [14], while the exploration of dynamic tori has been studied in [22] with the assumption that each ring in the torus is constantly connected. The problem of gathering many agents at a node [15] or periodically patrolling the nodes using many agents [13] has been studied for constantly connected dynamic ring networks (See the recent survey by Di Luna [29] for many of the above results). The only previous work in this model which considers arbitrary topologies is by Balev et al. [3] who studied the problem of *cops and robbers* on sparse graphs of arbitrary topology. In this problem, a team of agents called cops have to capture (i.e., meet) a malicious agent called the robber, while the cops and the robber move in alternate rounds (the adversary may change the graph after both have moved). The similarity of this problem with the broadcast problem is only the first step when one of the ignorant agents needs to meet the source agent (although they can collaborate unlike in the cops and robbers problem). One major difference between the problems is that the cops can choose their location in the graph, unlike the agents in our problem that are placed by the adversary. Consequently the approach used in [3] cannot be adapted to our problem. Moreover the bounds on the team size even for simple topologies are different for the two problems. Only in the trivial case of tree networks (when the graph is essentially static) both problems allow a solution with a single ignorant agent (cop).

In static networks, the communication problems of broadcast among mobile agents were studied by Anaya et al. [1] for agents with limited energy, while Czyzowicz et al. [8, 9] studied the problem with the objective of energy optimization. Other types of faults that have been considered for static networks are faulty agents in the context of collaborative patrolling [10] or the presence of a mobile adversary that blocks the path of agents [11, 12].

## Our Contributions

In this paper we consider the broadcast problem for mobile agents in dynamic networks with various underlying topologies. For each topology, we determine the minimum number of agents that makes the problem solvable, which apparently depends on the density of

the underlying graph  $G$  or, the number of redundant edges in  $G$ . For tree networks, the adversary can never delete an edge without disconnecting the graph, thus broadcast is always solvable for any number of agents. When the underlying graph is a ring, we show that at least 2 agents (apart from the source agent), are needed, except for small rings of less than 5 nodes. For cactus graphs, which are collections of rings that can pairwise intersect in no more than one node, we show that the number of agents necessary must be more than the number of cycles. We then consider denser and regular graphs. For grid graphs, where the total number of edges in  $G$  is still linear in  $n$ , we show that agent teams of size  $\Omega(n)$  are needed for broadcast. For the special case of grids with only two rows, we have tight results and a strategy for solving broadcast using one more than the minimum number of agents. For complete graphs, which are the densest graphs we show a tight result that  $k = n - 2$  agents are necessary and sufficient. Finally we consider hypercube networks, and we show that almost half of the nodes of the graph must be occupied to succeed in solving broadcast. For the special case of 3-dimensional hypercube, we show a tight result of  $k = n/2$  agents, while for higher dimensional hypercubes, there is a gap between the lower and upper bounds on the team size needed for solving broadcast.

The paper is organized as follows: In Section 2 we introduce the model and the required background, followed by some preliminary observations. In Section 3 we present solutions for sparse network topologies such as rings and cactus graphs. Section 4 considers grids graphs, while in Section 5 we present solutions for dense networks including hypercubes and complete graphs. To the best of our knowledge, these are the first results on broadcast with mobile agents in dynamic graphs, and unlike previous work in this model, we consider various distinct topologies, providing new techniques for dealing with the dynamicity in these networks.

## 2 The Model

### 2.1 The network and the agents

The network topology is given by a connected graph  $G = (V, E)$  with  $n = |V|$  nodes. The network is locally oriented in the following sense. All edges incident to a node have distinct port labels. However, the network is dynamic – not all edges are available at all times. We model the network as *Constantly connected* Dynamic Graphs denoted by  $\mathcal{G} = \{G_0 = (V, E_0), G_1 = (V, E_1), \dots\}$ , as a sequence of static graphs, where  $G_r$  corresponds to the graph at round  $r$ . We emphasize that apart from the restriction that the graph remains connected at any time, there is no other assumption or restriction with respect to which or how many links might fail at a time. For instance, an adversary might keep deactivated any number of links forever, as long as the graph remains connected. Thus, for any  $r > 0$ ,  $G_r$  is any connected spanning subgraph of the original graph  $G$ . The distance between nodes  $u$  and  $v$  in  $G$  is denoted by  $d_G(u, v)$ .

**The agents:** The agents are autonomous entities with distinct identifiers. Each agent has its own internal memory and is able to move along the available edges of the graph. The agents cannot leave marks on the nodes or the edges of the graph. The agents are initially located at distinct nodes of the network, they all start in the same initial state, and they execute the same deterministic algorithm. They have *global visibility* and they move in *synchronous* steps, i.e., time is discretized into atomic time units called rounds. During each round  $r$ , each agent can see the graph  $G_r$  and the location of all agents in  $G_r$  along with their identifiers, and can distinguish which agents have the message  $\mathcal{M}$  (called *source agents*), and which agents do not (called *ignorant agents*). Based on this information, the

agent can decide to stay at the current node or move to a neighboring node in  $G_r$ . In the latter case, the agent arrives at its destination node at the end of the round. At the start of the next round  $r + 1$ , the adversary chooses the graph  $G_{r+1}$ , and the agents execute the next step of the algorithm. In the initial round, there is exactly one source agent and  $k$  ignorant agents in the network.

**The adversary:** The adversary can decide the initial placement of all the agents in the network, and in each round the adversary chooses the graph  $G_r$  which represents the available links in the network for that round. The adversary may have knowledge of the algorithm and can use this knowledge for deciding the placement of agents and the dynamicity of the network (subject to the connectivity constraint as described).

**Unknown Adversary:** The agents do not have any knowledge of the adversary, and thus they do not know the dynamic network  $\mathcal{G}$  in advance.

As mentioned before, the adversary in this model is quite powerful, which makes it necessary to make some strong assumptions about the capabilities of the agents. In particular the global visibility makes it easier to coordinate among the agents as they all have the same knowledge about the network in each round. The assumption about distinct initial locations is not strictly necessary, since the agents could move to distinct nodes by virtue of their distinct identifiers. On the hand, if the agent do not have distinct identities but start from distinct locations, it is possible to assign distinct identifiers to the agents at the start of the computation, due to the global visibility assumption and the fact that there is a uniquely identifiable source agent initially. For simplifying the discussion, we assume agents have distinct identities and start at distinct locations. Moreover, we shall describe the algorithms in a centralized manner, describing which agents performs which operations in any round. It is evident that the agents executing the same algorithm, can autonomously decide their role in the computation.

## 2.2 Preliminaries

We make some preliminary observations about the problem.

► **Observation 1.** *Given a constantly connected dynamic network  $\mathcal{G}$  based on the underlying graph  $G$  consisting of  $n$  nodes,  $k \geq n - 2$  ignorant agents starting from distinct nodes of the network, can solve the broadcast problem.*

**Proof.** Since there are at least  $n - 2$  ignorant agents on distinct nodes and one node is occupied by the source agent, there is at most one empty node. Thus, in any connected graph  $G_i$  there would be a path of length at most two between a source agent and an ignorant agent. These two agents would meet in this round. So, we reduced the number of ignorant agents. The agents can now spread to distinct nodes by virtue of their distinct identities and we can repeat the same argument. ◀

The above result provides a general upper bound on the team size needed for solving broadcast. We will present smaller lower bounds for specific topologies. For the special case of trees, the adversary cannot block any edge with losing connectivity. Hence we have the following trivial upper bound for trees.

► **Observation 2.** *If  $G$  is a tree then broadcast can be solved for any  $k \geq 1$ .*

**Proof.** The adversary cannot block any edge without disconnecting the graph. Thus, the graph is static and in each step each agent can move one step closer to the node containing the source agent, thus in  $O(D)$  time all agents would be colocated with the source agent and we solve broadcast in  $O(D)$  time, where  $D$  is the distance of the farthest agent from the source. ◀

### 3 Broadcast in sparse graphs

In this section, we will study the broadcast problem with agents in sparse graphs. The simplest non-trivial network is the ring topology.

► **Theorem 3.** *If  $G$  is a ring of size  $n \geq 5$ , then broadcast can be solved if and only if  $k \geq 2$ . If  $G$  is a ring of size  $n < 5$ , then broadcast can be solved for any  $k \geq 1$ .*

**Proof.** Consider a ring of size  $n \geq 5$ , with one source that has the information  $\mathcal{M}$  and one ignorant agent. In each round, the adversary can remove an edge on the shortest path between the two agents. Note that, the longer path is always of size at least 3, thus the agents cannot meet on this path in this round. Hence the two agents can never meet.

Now we show that if there are at least two ignorant agents ( $k \geq 2$ ), then broadcast is possible. The two agents can try to reach the source agent by moving towards it from opposite directions, then at each step one of the agents gets closer, and eventually one of the agents would reach the source and obtain  $\mathcal{M}$ . At this stage there are 2 source agents, they can traverse the ring in opposite directions, thus at least one of them will soon meet the remaining ignorant agent and broadcast is solved.

The impossibility of solving broadcast with  $k = 1$ , does not hold for rings of size 3 as in this case any path between the source and the other agent is of size at most 2, and since one of these paths must be available, the two agents can meet in one step and solve the problem. For rings of size 4, if the longer path between the source agent and an ignorant agent has length 3, then one of the agents can always move such that both paths between the agents are of length 2. Then within the next step, the two agents meet in one of those paths, similarly as in the case of rings of size 3. Thus, broadcast is solvable for rings of size  $n < 5$  with any  $k \geq 1$ . ◀

We now make the following observation that will allow us to generalize the results from rings to other graphs containing cycles.

► **Lemma 4.** *If  $G$  contains a cycle  $C$  of length at least 3, such that there is a single node  $v \in C$  that is connected to nodes in  $G \setminus C$ , then the adversary can always prevent at least one agent located in  $C \setminus v$  from reaching node  $v$ , thus trapping the agent in cycle  $C$ .*

**Proof.** Consider an agent located at a node  $u \in C \setminus v$ . Since the cycle must be of length at least 3, the longer path from node  $u$  to node  $v$  is of length  $\geq 2$ . If the adversary always blocks the shorter path from the agent's location to node  $v$ , then the agent cannot reach node  $v$ . The only way to get out of the cycle is passing through node  $v$ , so the agent is forever trapped in  $C$ . ◀

► **Lemma 5.** *If  $G$  is a ring of size  $n \geq 3$ , given any node  $v \in G$ , if there are two agents at distinct nodes of  $G$ , then there is an algorithm to ensure that within  $n$  rounds either (i) the two agents meet at a node of  $G$  or (ii) at least one of the agents (chosen by the algorithm) can reach node  $v$ .*

**Proof.** Let us call the two agents  $A$  and  $B$ . If we require agent  $A$  to reach node  $v$ , then the algorithm asks agent  $B$  to move along the path containing agent  $A$  and then node  $v$  in this order (if agent  $B$  was already at node  $v$  then we first move it to any neighbouring node of  $v$ ). In each round, only one edge of  $G$  may be unavailable, so either agent  $B$  will move closer to agent  $A$  or agent  $A$  will move closer to node  $v$ . So eventually either condition (i) or (ii) will be true. ◀



In the following we consider cactus graphs which can be seen as combinations of trees and rings.

► **Definition 6.** *A cactus graph is a connected graph in which any two simple cycles have at most one node in common.*

In cactus graphs, the size of the team depends on the number and sizes of the cycles of the graph, as follows:

► **Lemma 7.** *If  $G$  is a cactus graph of size  $n$  having  $c_1 \geq 1$  cycles of length  $< 5$  and no larger cycles, then broadcast can be solved if and only if  $k \geq c_1$ .*

**Proof.** Consider the family of cactus graphs obtained from a line of length  $c_1$  by attaching to each node of the line a cycle of length  $< 5$ . Each cycle thus satisfies the conditions of Lemma 4.

If  $k < c_1$ , the total number of agents is  $k + 1 \leq c_1$ , so the adversary can place each agent in a distinct cycle, including the source agent. No agent can leave its cycle due to Lemma 4. So, no two agents can meet, thus broadcast is not possible.

For  $k \geq c_1$ , broadcast is solvable in any cactus graph with  $c_1$  small cycles. To prove this it is enough to analyze the case where at least one cycle has at least two agents, either two ignorant ones or one ignorant agent and a source. This is because if an agent is outside of any cycle it can always move to a cycle (all non cyclic edges are available in each round). If two agents are in a cycle and none of them is the source, then one of the agents can leave the cycle within at most 2 steps (both agents try to move towards an elected exit by approaching it from different directions). The agent that leaves the cycle can move towards the source, until it reaches another cycle.

Thus, one agent will eventually reach the cycle containing the source. This agent can meet the source and obtain the information  $\mathcal{M}$  due to Theorem 3. Now, there are two source agents in the same cycle, and thus, one of them can leave the cycle as described before within at most 3 steps. This source agent reaches another ring containing an ignorant agent, the information is propagated and we have again two source agents in a cycle. Repeating the same algorithm, all agents will eventually learn the information, and thus we can solve broadcast. ◀

► **Lemma 8.** *If  $G$  is a cactus graph of size  $n$  having  $c_2$  cycles of length  $\geq 5$  and no cycles of smaller length, then broadcast can be solved if and only if  $k \geq c_2 + 1$ .*

**Proof.** Consider the family of cactus graphs obtained from a line of length  $c_2$  by attaching to each node of the line a cycle of length  $\geq 5$ .

Suppose that  $k < c_2 + 1$ . Then, the adversary places each of the  $k \leq c_2$  agents in a distinct cycle and the source in one of these cycles. Due to Theorem 3, the source and the other agent cannot meet, since the ring is of length  $\geq 5$ . At the same time, due to Lemma 4 the adversary can trap each other agent in its cycle. In the cycle that contains the source, at most one of the two agents can exit this cycle. Even if the source agent exits this cycle and enters another cycle the configuration is similar to the initial one. Hence, no two agents can ever meet and therefore the problem is unsolvable.

To prove that  $k \geq c_2 + 1$  agents are enough to solve broadcast, we first show that at least one ignorant agent can reach the source and can become a new source. As in the proof of Lemma 7, we assume all agents move to some cycle if they are not in a cycle. If the source agent is in the same cycle with at least two ignorant agents, by Theorem 3 both these agents can become sources. If not, then, given  $k \geq c_2 + 1$ , there must be some cycle with two or

more ignorant agents and all except one of these agents can leave the cycle to reach another cycle. Eventually two or more ignorant agents would reach the same cycle as the source, and again applying Theorem 3, all these agents would become source agents. Thus we have now at least  $x \geq 3$  source agents in a cycle. Furthermore each of the remaining  $k - x + 1$  ignorant agents are alone in some cycle. This implies that the number of empty cycles (cycles without any agent) are at most  $x - 3$ . Among the  $x$  source agents,  $x - 1$  of them can move to another cycle. Whenever these source agents arrive at an empty cycle, at most one of them may be trapped. In total  $x - 3$  source agents can be trapped, thus at least two source agents can reach any other cycle that contains an ignorant agent, so this agent will meet a source. Hence, all ignorant agents will eventually become sources and thus broadcast can be solved. ◀

► **Theorem 9.** *Let  $G$  be a cactus graph of size  $n$  having  $c_1$  cycles of length  $< 5$  and  $c_2$  cycles of length  $\geq 5$ , then:*

- *If  $c_2 = 0$ , broadcast can be solved if and only if  $k \geq c_1$ .*
- *If  $c_2 > 0$ , broadcast can be solved if and only if  $k \geq c_1 + c_2 + 1$ .*

**Proof.** If  $c_2 = 0$ , then the cactus graph has only  $c_1$  cycles of length  $< 5$ , and in view of Lemma 7 broadcast can be solved if and only if  $k \geq c_1$ .

On the other hand if  $c_2 > 0$  and  $c_1 = 0$ , then in view of Lemma 8 broadcast can be solved if and only if  $k \geq c_2 + 1$ , and thus the second condition holds.

Finally, if  $c_1 > 0$  and  $c_2 > 0$ , similarly as in the proofs of Lemmas 7 and 8, we can construct a cactus graph from a line of length  $c_1 + c_2$  by attaching a cycle of length  $< 5$  to each of the first  $c_1$  nodes and attaching a cycle of length  $\geq 5$  to each of the remaining  $c_2$  nodes. Now, if  $k \leq c_1 + c_2$ , then the adversary places each ignorant agent in a distinct cycle, and places the source agent in the last big cycle  $C$  of length  $\geq 5$ . Since there is at most one ignorant agent and the source in cycle  $C$  of length  $\geq 5$ , they cannot meet (see the proof of Theorem 3). Furthermore no other agent (in a cycle different than  $C$ ) can leave its cycle due to Lemma 4. The source agent may escape from the cycle  $C$  and reach another cycle  $C'$ . If the cycle  $C'$  is big (size  $\geq 5$ ), then as before the source agent would not be able to meet the only agent that is in cycle  $C'$ . On the other hand, if the source reaches a small cycle (size  $< 5$ ) it may meet the ignorant agent in that cycle, so we will have two sources; however at most one of the two can leave this cycle. Thus the agents in the big cycles would never meet any source agent. Thus broadcast can not be solved.

We now show how to solve broadcast using  $k \geq c_1 + c_2 + 1$  ignorant agents. First, as argued before, any agent that is not on a cycle can move to the nearest cycle. Since there are more agents than cycles, there are some cycles that contain multiple agents. In any such cycle, one of the agents can move to a neighboring empty cycle if there is one. Repeating this process, we can distribute the agents such that there is at least one agent in each cycle.

Let  $C$  be the cycle that contains the source. In any cycle other than  $C$ , if there are more than one ignorant agents, all except one of them can move to another cycle that is closer to cycle  $C$ . Repeating this process, we will reach a configuration where there will be at least 2 ignorant agents and the source in cycle  $C$  and exactly one agent in each other cycle. Now it is easy to solve broadcast from this configuration. Using the ring algorithm (Theorem 3) all ignorant agents in cycle  $C$  would become sources. Since we have at least three source agents now, at least two of them can move to a different cycle. In any other cycle reached by those two source agents, there is one ignorant agent, so we can apply the same algorithm and have 3 source agents in this cycle. Repeating this process all ignorant agents will become sources and broadcast is solved. ◀

## 4 Broadcast in Grids

We now study grid graphs which are slightly more dense than rings or cactuses. Even for 2-dimensional grids, we show that we need  $\Omega(n)$  agents to solve broadcast. Let us first consider the simplest grid graph with only two rows (so called ladder graph).

► **Theorem 10.** *If  $G$  is  $2 \times L$  grid graph, then broadcast is unsolvable for  $k < L - 1$ .*

**Proof.** Suppose that  $k < L - 1$ . The adversary can put all  $k$  agents in one row consisting of  $L$  nodes and the source agent in the other row. So, there is at least one column where both nodes are empty. The adversary would allow this edge and remove all other edges connecting the two rows. So the agents could only move within their respective rows. After some agents move, there would again be some column containing only empty nodes. So the above argument can be repeated. Thus, no agent can leave its respective row at any step, and hence no agent can meet the source. ◀

The above lower bound is almost tight as we can show an upper bound of  $k \geq L$  for broadcast in any  $2 \times L$  grid graph.

► **Theorem 11.** *If  $G$  is a  $2 \times L$  grid graph, then broadcast is solvable for  $k \geq L$ .*

The proof is omitted, but we present some basic ideas here. The algorithm for broadcast in  $2 \times L$  grids is based on the following lemma which ensures that the ignorant agents can get closer and closer to the source until one of the agents meets the source.

► **Lemma 12.** *If  $G$  is  $2 \times L$  grid graph containing a source agent at origin  $[0, 0]$ , such that the number of nodes occupied by ignorant agents is strictly greater than the number of unoccupied nodes in  $G$ , then there exists a move of a subset of the agents which maintains the ignorant agents in distinct locations and, either (i) one ignorant agent meets the source agent, or (ii) the sum of distances from the ignorant agents to the source agent in  $G$ , decreases by at least one.*

To see why the lemma holds, consider any agent  $x$  and the path from this agent to the source during the current round of the algorithm (there always exists such a path as the dynamic graph is constantly connected). If all the agents on this path simultaneously move by one edge along this path, and if there are more agents than unoccupied nodes, then we can show more than half of the agents would get closer<sup>1</sup> to the source. Thus the sum of distances from the agents to the source will decrease monotonically and the algorithm makes progress. Note that there are  $k \geq L$  agents in distinct locations, there are at most  $L - 1$  unoccupied nodes. Thus, there is always some path with more occupied nodes than unoccupied nodes. However the algorithm should always maintain the agents in distinct locations. There are several other technical difficulties that need to be overcome to make the algorithm work, for example, when the source is not in the corner node but somewhere in the interior then we need to partition the grid and apply the lemma selectively on one partition and so on. The resulting algorithm is quite involved and the above approach does not generalize to general grids with more than two rows.

Indeed in larger grids we need a larger team of agents to solve the problem. We will now consider the general case of a  $W \times L$  grid graph for  $W, L > 2$  and show some lower bounds for broadcast in such grids. We first prove the following technical lemma.

---

<sup>1</sup> closer in terms of distances in the original graph  $G$

## 24:10 Broadcasting with Mobile Agents

► **Lemma 13.** *If  $G$  is an  $h \times L$  grid graph, with  $h \geq 1, L > 2$ , then starting from a configuration with  $L - 1$  agents in each row, the adversary can ensure that there are never more than  $L - 1$  agents in the bottom row. Moreover, if we add an additional agent at the bottom row, again the adversary can ensure that there are never more than  $L$  agents in the bottom row.*

**Proof.** If  $h = 1$  then there is only one row, so the number of agents on the bottom row never changes and the lemma holds trivially. We now prove the lemma by induction on the number of rows. Suppose that the lemma holds for a grid  $G$  with  $h \geq 1$  rows. We can construct a grid  $G'$  by adding an additional row at the bottom with  $L$  nodes and  $L - 1$  agents on distinct nodes. Since the bottom row of  $G$  has at most  $L - 1$  agents (by induction hypothesis), there exists an empty node  $v$  in this row. In the grid  $G'$  the adversary makes available the edge from  $v$  to the node directly below (on the additional row); all other edges between  $G$  and the additional row are unavailable. In the current round, no agent can enter the bottom row of  $G'$ ; although an agent from the bottom row may go up and reach grid  $G$ . In the next round if there are  $L - 2$  agents in the bottom row, and  $L$  agents in the row above then the adversary makes available the edge between a node (containing at most one agent) in this row to the node below in the bottom row (and disables all other edges between these two rows). So, either an agent moves to the bottom row or the number of agents in the bottom row remains the same. In the first case, we are back in the initial situation and we could use the same arguments as before. In the second case, the number of agents at the bottom row is smaller than what was initially. Thus the lemma holds for grids of  $h + 1$  rows and thus by induction for all grids satisfying the conditions of the lemma. Furthermore, note that all arguments remain the same if initially there are  $x > L - 1$  agents in the bottom row, i.e., the number of agents in the bottom row is never more than  $x$  if the higher rows contain at most  $L - 1$  agents initially. ◀

► **Lemma 14.** *If  $G$  is a  $W \times L$  grid graph, with  $W > 2$  then broadcast is unsolvable for  $k < (L - 1)(W - 1)$ .*

**Proof.** We can construct a grid of size  $W \times L$  by joining two grids: a grid  $G^1$  of  $(h = W - 2) \times L$  with a grid  $G^2$  of size  $2 \times L$  (by adding  $L$  edges between the bottom row of  $G^1$  and the top row of  $G^2$ ). In grid  $G^1$ , we place  $L - 1$  ignorant agents on each row, at distinct locations, while in grid  $G^2$  we place  $k_2 < (L - 1)$  agents plus the source agent, as in Theorem 10. Thus the total number of agents is  $k = (W - 2) * (L - 1) + k_2 < (L - 1)(W - 1)$ . We now show that broadcast is not solvable in this graph.

First, if no additional agents enter grid  $G^2$  and the source never leaves  $G^2$  then by Theorem 10, broadcast is not possible as no agent would meet the source in  $G^2$ . Furthermore, if the source is in the bottom row, it can never leave this row and thus it cannot leave  $G^2$ . In the grid  $G^1$  there are  $L - 1$  agents in the bottom row and by the Lemma 13 the number of agents on this row does not increase (considering only agents in  $G^1$ ), so there is at least one empty node on this row. The edge between this node and the grid  $G^2$  is made available and all other edges between the two grids are unavailable. In this case, no agents from  $G^1$  can enter  $G^2$ . However, some ignorant agents from  $G^2$  can enter  $G^1$ . If  $x$  ignorant agents from  $G^2$  enter the grid  $G^1$ , then by Lemma 13, the number of agents on the bottom row of  $G^1$  is at most  $L - 1 + x$ . In each round, if there is some empty node  $v$  on this row, the edge between  $v$  and  $G^2$  is the only edge between the two grids that is made available in that round (in this case, no agents can move from  $G^1$  to  $G^2$ ). Otherwise any node in the bottom row of  $G^1$  can have at most  $x$  agents in this round; so, if one edge is available between the two grids, at most  $x$  agents can move from  $G^1$  to  $G^2$ . Thus, after each round, the number of ignorant agents in  $G^2$  is less than  $L - 1$  and thus broadcast is not possible by Theorem 10. ◀

We can generalize the above to higher dimensional grids as follows:

► **Theorem 15.** *If  $G$  is a  $(d+1)$ -dimensional grid graph of size  $W_1 \times W_2 \times \dots \times W_d \times L$  where  $W_i \geq L \geq 2$  then broadcast is unsolvable for  $k < (L-1)(W_1 \cdot W_2 \dots W_d - 1)$ .*

**Proof.** Consider the subgraph of the grid that is a 2-dimensional grid of size  $(W_1 \cdot W_2 \dots W_d) \times L$ . In each round the adversary chooses the available graph to be a connected subgraph of this 2D grid, then we can apply Lemma 14 to obtain the above lower bound. ◀

## 5 Broadcast in Dense graphs

In dense graphs there are many disjoint paths between two nodes and thus there are many possible ways for the adversary to change the network while keeping it connected. In other words, the dynamicity of a (constantly connected) dynamic graph whose footprint is a dense graph, is higher than that of sparser graphs that we studied before. The worst case is when the underlying graph is a complete graph.

### 5.1 Broadcast in Complete graphs

► **Theorem 16.** *If  $G$  is a complete graph of size  $n$  then broadcast can be solved if and only if  $k \geq n - 2$  within  $O(n)$  steps.*

**Proof.** If  $k < n - 2$  then at most  $n - 2$  nodes are occupied (including the source), and therefore there are at least two empty nodes. The adversary will make available the spanning tree where the source is connected to one empty node and all other occupied nodes are connected to the other empty node. The two empty nodes are connected with an edge. This is a spanning tree of  $G$  and in this tree, the distance from the source to any occupied node is more than two. So no agent can meet the source in one step. After one step, during which some agents may move, there will still be at least two empty nodes; thus the same argument can be repeated for any step. Hence broadcast is impossible for  $k < n - 2$ .

If  $k \geq n - 2$ , then by the Observation 1, it is possible to solve broadcast in any arbitrary topology, and thus in a complete graph. ◀

The impossibility result above can be generalized to arbitrary graphs  $G$  having the following property.

► **Lemma 17.** *Consider a graph  $G$  and an integer  $k \geq 1$ . Suppose that for every possible placement of the source agent and  $k$  agents on distinct nodes of  $G$ , there always exists a spanning tree of  $G$  where the distance from each agent to the source is  $\geq 3$ . Then, broadcast is impossible in  $G$  with  $k$  ignorant agents.*

### 5.2 Broadcast in Hypercubes

We now study the problem in hypercube networks as defined below.

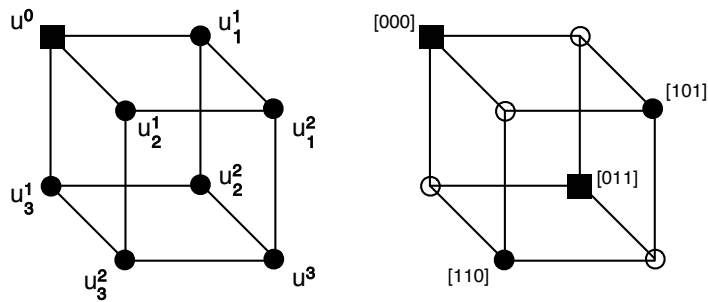
► **Definition 18.** *A  $d$ -dimensional hypercube is a graph  $H_d = (V, E)$  with  $n = 2^d$  nodes labelled with distinct  $d$ -bit strings. A node  $v_i \in V$ ,  $0 \leq i \leq 2^d - 1$ , is connected to the  $d$  nodes whose labels differ in exactly one bit from its own. Hereafter, we freely identify nodes with their labels.*

A hypercube  $H_d$  consists of two  $d - 1$  dimensional hypercubes labelled as  $[0 * * \dots *]$  and  $[1 * * \dots *]$ , the corresponding nodes of these two hypercubes are connected by edges of dimension  $d$ .

► **Theorem 19.** *Given a hypercube  $H_d$  of dimension  $d > 2$ , at least  $k = n/2 - 1$  ignorant agents are necessary to solve broadcast in the dynamic graph based on  $H_d$ .*

**Proof.** Assume  $k < n/2 - 1$ . Suppose the source agent is at the node  $[00\dots 0]$  of  $H_d$ ; the adversary places all the ignorant agents among the nodes of the sub-hypercube  $H_{d-1}$  labelled  $[1**\dots*]$ . Out of the  $n/2$  nodes in  $H_{d-1}[1**\dots*]$ , there must be at least 2 empty nodes. At most one of these two nodes can be a neighbor of the source node  $[00\dots 0]$  in the other sub-hypercube. So, the other empty node  $v$  must be a neighbor of an empty node  $u$  in  $H_{d-1}[0**\dots*]$ . The adversary chooses the available graph  $G_i$  as the union of a spanning tree of  $H_{d-1}[0**\dots*]$  and a spanning tree of  $H_{d-1}[1**\dots*]$ , plus the edge  $(u, v)$ . All other edges of dimension  $d$  are missing in  $G_i$ . After the agents move in this round, the ignorant agents would still be in sub-hypercube  $H_{d-1}[1**\dots*]$  and the source would be in the other sub-hypercube  $H_{d-1}[0**\dots*]$ . Thus, using the same argument, in each round  $r$  the adversary can choose the graph  $G_r$  as a spanning tree where each ignorant agent is at a distance of at least 3 from the source, so by Lemma 17, broadcast is impossible. ◀

The above result does not hold for the trivial case of  $d = 2$ , since  $H_2$  is simply a ring of four nodes where broadcast can be solved even for  $k = 1$  (see Theorem 3). For a hypercube of dimension  $d = 3$  (i.e., a cube) we can show a matching lower and upper bound of  $k = n/2$  ignorant agents.



■ **Figure 1** (a) The cube with a single source (denoted by a square) in the proof of Theorem 20. (b) The cube with two sources at distance 2; the remaining agents must occupy the two black nodes.

► **Theorem 20.** *If  $G = H_3$  is a hypercube of dimension  $d = 3$  consisting of  $n = 2^3$  nodes, then  $k = n/2$  ignorant agents are necessary and sufficient to solve broadcast.*

**Proof (Lower Bound).** We provide only a sketch of the proof that if there are only  $k = 3$  ignorant agents, then it is not possible to solve broadcast starting from arbitrary configurations. In particular, we define a class  $QF$  of forbidden initial configurations, with one source and 3 ignorant agents in a cube, such that starting from any such configuration, the adversary can force the agents to move only to another configuration in  $QF$ . Further, in every configuration in  $QF$ , there is a spanning tree (defined by the available links) where the distance from source to the nearest agent is at least 3. The configurations in the set  $QF$  are listed below by showing the positions of the 3 agents, with respect to the source node which is always assumed<sup>2</sup> to be  $[000]$ :

<sup>2</sup> After any moves, we rename the nodes.

**Q1** ([100], [010], [110])

**Q2** ([100], [010], [011])

**Q3** ([100], [110], [101])

**Q4** ([100], [101], [011])

**Q5** ([100], [110], [111])

**Q6** ([100], [011], [111])

**Q7** ([101], [011], [111])

Note that starting from any other initial configurations with 3 agents permits a solution to broadcast. However, when the 3 ignorant agents start in any configuration isomorphic to configurations in  $QF$ , then the configuration in the next round can only be another configuration in  $QF$ . This implies that no ignorant agent can meet the source after any number of rounds, and thus it is not possible to solve the Broadcast problem. ◀

**Proof (Upper Bound).** We show that  $k = 4$  agents can solve broadcast by providing an algorithm. We first show that one of the four agents can meet the source agent. We denote the nodes of the cube as follows:  $u^0 = [000]$  is the node containing the source,  $u_1^1, u_2^1, u_3^1$  are the three nodes at distance one from source,  $u_i^1$  means a generic node at distance 1 from the source,  $u_1^2, u_2^2, u_3^2$  are the three nodes at distance 2 from the source, and  $u^3 = [111]$  is the only node at distance 3 (see Figure 1(a)). We now consider all possible initial configurations with agents placed on distinct nodes; Note that, each such configuration has at least three empty nodes. We denote such a configuration as  $[x, y, z, l]$  showing the positions of the 4 ignorant agents at nodes  $x, y, z, l$  (with  $*$  denoting any node other than the source).

**C1** Configuration  $[u_1^1, u_2^1, u_3^1, *]$ : At least one of the links  $(u^0, u_1^1)$ ,  $(u^0, u_2^1)$ , or  $(u^0, u_3^1)$  must be active, otherwise node  $u^0$  would be disconnected. Hence at least one agent can meet the source within the next time unit.

**C2** Configuration  $[u_1^2, u_2^2, u_3^2, *]$ : At least one of the paths of distance two between the source node  $u^0$  and one of the nodes  $u_1^2, u_2^2$ , or  $u_3^2$  must be available, otherwise node  $u^0$  would be disconnected from nodes  $u_1^2, u_2^2$  and  $u_3^2$ . Hence, within the next step the agent at a distance two from the source, and the source agent move to the middle node of the path and meet.

**C3** Configuration  $[u_1^1, u_2^1, u^3, *]$ : If at least one of the links  $(u^0, u_1^1)$  or  $(u^0, u_2^1)$  are active, then at least one agent can meet the source within the next time unit. Otherwise, the link  $(u^0, u_3^1)$  must be active (to ensure connectedness), so in that case, the source agent moves to node  $u_3^1$ . The configuration we obtain is isomorphic to the configuration  $[c_2]$  above, so we are done.

**C4** Configuration  $[u_1^1, u_2^1, u_1^2, u_2^2]$ : Assume that there are no paths of length 1 or 2 from source to any agent (otherwise we are done as explained above). In that case, any possible spanning tree must have the edges  $(u^0, u_3^1)$  and  $(u_3^1, u_3^2)$  and further at least one of the edges  $(u_2^1, u_3^2)$  or  $(u_1^2, u_3^2)$  or  $(u_2^2, u_3^2)$ . In the first case, one agent moves to  $u_3^2$  and we obtain the configuration  $[c_2]$ . In the other two cases, one agent moves to node  $u^3$ , and thus we obtain the configuration  $[C3]$ . So, we are done in all cases.

**C5** Configuration  $[u_i^1, u_1^2, u_2^2, u^3]$ : In  $G_0$ , if there are no length-2 paths from source to any agent, then any path from  $u^0$  must go through  $u_3^2$ ; If node  $u_3^2$  has an available edge to some agent, this agent will move to  $u_3^2$  and we would obtain the configuration  $[c_2]$ . Otherwise  $u_3^2$  has an available edge to some empty node  $u_j^1 \neq u_i^1$  which is connected to some node occupied by an agent. Thus, this agent moves to  $u_j^1$ , and we obtain the configuration  $[C3]$ .



We have shown one agent can reach the source within a constant number of steps and obtain the message. Now, there are two source agents and three ignorant agents. The two source agents can place themselves at distance two in  $G$  (this is always possible in at most 2 steps). Assume without loss of generality, that the two sources are at nodes  $[000]$  and  $[011]$  as in Figure 1(b). There are exactly two nodes ( $[110]$  and  $[101]$ ) that are at distance 2 from both the sources. If the ignorant agents occupy these two nodes, then in any spanning tree chosen by the adversary, at least one of the ignorant agents would be at distance two from a source agent. And in fact, it is easy to see that either the three ignorant agents occupy all three adjacent nodes of one of the two sources (which means that in the next step at least one more agent will meet a source), or three ignorant agents occupy two adjacent nodes of each source. In the last case (due to connectedness) at least two of the ignorant agents will occupy the two nodes ( $[110]$  and  $[101]$ ) in the next step. Finally, if only one of the ignorant agents occupies one of the nodes ( $[110]$  and  $[101]$ ), then in the next step this agent can move from that node and therefore we obtain the previous configuration (i.e., where three ignorant agents occupy two adjacent nodes of each source). Thus eventually at least two of the ignorant agents will occupy the two nodes ( $[110]$  and  $[101]$ ) and in the next round, at least one more ignorant agent will meet a source and therefore we will have 3 source agents.

Note that the case of 3 sources and 2 ignorant agents is analogous to the case of 2 sources and 3 ignorant agents, while the case of one ignorant agent and four sources, is analogous to the initial situation with one source and four ignorant agents. So, using the same strategies as above eventually all agents will obtain the message and broadcasting is solved. ◀

For hypercubes of higher dimensions  $d \geq 4$ , we do not have any general strategy for solving the problem as the adversary has too many possible ways of choosing the available subgraph. However, the lower bound of  $k = n/2 - 1$  agents from Theorem 19 still holds and we have the upper bound of  $k = n - 2$  (from Observation 1).

## 6 Conclusion

In this paper, we studied the problem of broadcast for mobile agents moving in constantly connected dynamic networks. The main objective is to understand how many agents are necessary and sufficient to allow broadcast to be solved in various topologies. It turns out that for sparse topologies such as rings and cactus graphs, the number of agents needed for solving the broadcast problem can be independent of the network size  $n$ , while for denser graphs including grids, hypercubes, as well as the complete graph,  $\Theta(n)$  agents are needed. This preliminary investigation on broadcast in dynamic graphs opens many new research directions. For both grids and hypercubes, we have large gaps between the lower bounds of  $(n - 2\sqrt{n})$  and  $(n/2 - 1)$  respectively, and the upper bound of  $(n - 2)$ . It seems that solving the problem in grids requires more agents than in hypercubes, since grid networks contain more redundant edges. However, the lower bound on hypercubes shows that the number of agents needed can sometimes be much more than the number of redundant edges in the network. This is in contrast to the *cops and robbers* problem where the number of cops needed is roughly equal to the number of redundant edges in the underlying graph [3]. In the future, we would like to study the differences between various problems in this model and try to adapt techniques used for broadcast, to solve other problems in dynamic networks. Moreover it would be nice to classify various problems according to the resources needed for solving them under the adversarial model studied in this paper. Another possible direction of research would be to replace the strong assumption of global visibility with some weaker assumptions about the agent's capabilities that still suffices to solve broadcast in this model.

## References

- 1 J. Anaya, J. Chalopin, J. Czyzowicz, A. Labourel, A. Pelc, and Y. Vaxès. Convergecast and broadcast by power-aware mobile agents. *Algorithmica*, 74(1):117–155, 2016.
- 2 B. Awerbuch and S. Even. Efficient and reliable broadcast is achievable in an eventually connected network. In *Proceedings of the 3th Symposium on Principles of Distributed Computing (PODC)*, pages 278–281, 1984.
- 3 S. Balev, J. J. Laredo, I. Lamprou, Y. Pigné, and E. Sanlaville. Cops and robbers on dynamic graphs: Offline and online case. In *Proc. Structural Information and Communication Complexity - 27th International Colloquium, SIROCCO 2020*, volume 12156 of *LNCS*, pages 203–219. Springer, 2020.
- 4 A. Casteigts, P. Flocchini, B. Mans, and N. Santoro. Shortest, fastest, and foremost broadcast in dynamic networks. *International Journal of Foundations of Computer Science*, 25(4):499–522, 2015.
- 5 A. Casteigts, P. Flocchini, W. Quattrociocchi, and N. Santoro. Time-varying graphs and dynamic networks. *International Journal of Parallel, Emergent and Distributed Systems*, 27(5):387–408, 2012.
- 6 M. Chrobak, L. Gasieniec, and W. Rytter. Fast broadcasting and gossiping in radio networks. *J. Algorithms*, 43(2):177–189, 2002.
- 7 A. Clementi, A. Monti, F. Pasquale, and R. Silvestri. Information spreading in stationary markovian evolving graphs. *IEEE Transactions on Parallel and Distributed Systems*, 22(9):1425–1432, 2011.
- 8 J. Czyzowicz, K. Diks, J. Moussi, and W. Rytter. Broadcast with energy-exchanging mobile agents distributed on a tree. In *Structural Information and Communication Complexity - 25th International Colloquium (SIROCCO)*, pages 209–225, 2018.
- 9 J. Czyzowicz, K. Diks, J. Moussi, and W. Rytter. Energy-optimal broadcast and exploration in a tree using mobile agents. *Theoretical Computer Science*, 795:362–374, 2019.
- 10 J. Czyzowicz, L. Gasieniec, A. Kosowski, E. Kranakis, D. Krizanc, and N. Taleb. When patrolmen become corrupted: Monitoring a graph using faulty mobile robots. *Algorithmica*, 79(3):925–940, 2017.
- 11 S. Das, R. Focardi, F. L. Luccio, E. Markou, and M. Squarcina. Gathering of robots in a ring with mobile faults. *Theoretical Computer Science*, 764:42–60, 2019.
- 12 S. Das, N. Giachoudis, F. L. Luccio, and E. Markou. Gathering of robots in a grid with mobile faults. In *45th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2019)*, volume 11376 of *LNCS*, pages 164–178. Springer, 2019.
- 13 S. Das, G. A. Di Luna, and L. A. Gasieniec. Patrolling on dynamic ring networks. In *Proc. 45th Int. Conf. on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, volume 11376 of *LNCS*, pages 150–163. Springer, 2019.
- 14 G.A. Di Luna, S. Dobrev, P. Flocchini, and N. Santoro. Live exploration of dynamic rings. In *Proceedings of the 36th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 570–579, 2016.
- 15 G.A. Di Luna, P. Flocchini, L. Pagli, G. Prencipe, N. Santoro, and G. Viglietta. Gathering in dynamic rings. In *Proceedings of the 24th International Colloquium Structural Information and Communication Complexity (SIROCCO)*, pages 339–355, 2017.
- 16 T. Erlebach, M. Hoffmann, and F. Kammer. On temporal graph exploration. In *Proceedings of 42nd International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 444–455, 2015.
- 17 A. Ferreira. Building a reference combinatorial model for manets. *IEEE Network*, 18(5):24–29, 2004.
- 18 P. Flocchini, B. Mans, and N. Santoro. On the exploration of time-varying networks. *Theoretical Computer Science*, 469:53–68, January 2013. doi:10.1016/j.tcs.2012.10.029.
- 19 L. Gasieniec. *Deterministic Broadcasting in Radio Networks*, pages 233–235. Springer US, Boston, MA, 2008. doi:10.1007/978-0-387-30162-4\_105.

- 20 L. Gasieniec and A. Pelc. Adaptive broadcasting with faulty nodes. *Parallel Computing*, 22(6):903–912, 1996.
- 21 L. Gasieniec and A. Pelc. Broadcasting with linearly bounded transmission faults. *Discrete Applied Mathematics*, 83(1-3):121–133, 1998.
- 22 T. Gotoh, Y. Sudo, F. Ooshita, H. Kakugawa, and T. Masuzawa. Group exploration of dynamic tori. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 775–785, July 2018.
- 23 F. Harary and G. Gupta. Dynamic graph models. *Mathematical and Computer Modelling*, 25(7):79–88, 1997.
- 24 D. Ilcinkas, R. Klasing, and A.M. Wade. Exploration of constantly connected dynamic graphs based on cactuses. In *Proceedings 21st International Colloquium Structural Information and Communication Complexity (SIROCCO)*, pages 250–262, 2014.
- 25 D. Ilcinkas and A.M. Wade. On the power of waiting when exploring public transportation systems. In *Proceedings of the 15th International Conference on Principles of Distributed Systems (OPODIS)*, pages 451–464, 2011.
- 26 D. Ilcinkas and A.M. Wade. Exploration of the t-interval-connected dynamic graphs: the case of the ring. *Theory of Computing Systems*, 62(5):1144–1160, 2018.
- 27 F. Kuhn, N. Lynch, and R. Oshman. Distributed computation in dynamic networks. In *Proceedings of the 42nd Symposium on Theory of Computing (STOC)*, pages 513–522, 2010.
- 28 F. Kuhn and R. Oshman. Dynamic networks: Models and algorithms. *SIGACT News*, 42(1):82–96, 2011.
- 29 G. A. Di Luna. Mobile agents on dynamic graphs. In *Distributed Computing by Mobile Entities, Current Research in Moving and Computing*, pages 549–584. Springer, 2019.
- 30 O. Michail. An introduction to temporal graphs: An algorithmic perspective. *Internet Mathematics*, 12(4):239–280, 2016.
- 31 O. Michail and P.G. Spirakis. Traveling salesman problems in temporal graphs. *Theoretical Computer Science*, 634:1–23, 2016. doi:10.1007/978-3-662-44465-8\_47.
- 32 R. O’Dell and R. Wattenhofer. Information dissemination in highly dynamic graphs. In *Proceedings of the Joint Workshop on Foundations of Mobile Computing (DIALM-POMC)*, pages 104–110, 2005.
- 33 D. Peleg and A. A. Schäffer. Time bounds on fault-tolerant broadcasting. *Networks*, 19(7):803–822, 1989. doi:10.1002/net.3230190706.
- 34 Y. Yamauchi, T. Izumi, and S. Kamei. Mobile agent rendezvous on a probabilistic edge evolving ring. In *2012 Third International Conference on Networking and Computing*, pages 103–112, December 2012.

# On Broadcast in Generalized Network and Adversarial Models

Chen-Da Liu-Zhang

Department of Computer Science, ETH Zürich, Switzerland  
lichen@inf.ethz.ch

Varun Maram

Department of Computer Science, ETH Zürich, Switzerland  
vmaram@inf.ethz.ch

Ueli Maurer

Department of Computer Science, ETH Zürich, Switzerland  
maurer@inf.ethz.ch

---

## Abstract

Broadcast is a primitive which allows a specific party to distribute a message consistently among  $n$  parties, even if up to  $t$  parties exhibit malicious behaviour. In the classical model with a complete network of bilateral authenticated channels, the seminal result of Pease et al. [10] shows that broadcast is achievable if and only if  $t < n/3$ . There are two generalizations suggested for the broadcast problem – with respect to the adversarial model and the communication model. Fitzi and Maurer [5] consider a (non-threshold) *general adversary* that is characterized by the subsets of parties that could be corrupted, and show that broadcast can be realized from bilateral channels if and only if the union of no three possible corrupted sets equals the entire set of  $n$  parties. On the other hand, Considine et al. [3] extend the standard model of bilateral channels with the existence of  $b$ -minicast channels that allow to locally broadcast among any subset of  $b$  parties; the authors show that in this enhanced model of communication, secure broadcast tolerating up to  $t$  corrupted parties is possible if and only if  $t < \frac{b-1}{b+1}n$ . These generalizations are unified in the work by Raykov [9], where a tight condition on the possible corrupted sets is presented such that broadcast is achievable from a complete set of  $b$ -minicasts.

This paper investigates the achievability of broadcast in *general networks*, i.e., networks where only some subsets of minicast channels may be available, thereby addressing open problems posed in [8, 9]. To that end, we propose a hierarchy over all possible general adversaries, and identify for each class of general adversaries 1) a set of minicast channels that are necessary to achieve broadcast and 2) a set of minicast channels that are sufficient to achieve broadcast. In particular, this allows us to derive bounds on the amount of  $b$ -minicasts that are necessary and that suffice towards constructing broadcast in general  $b$ -minicast networks.

**2012 ACM Subject Classification** Theory of computation → Cryptographic protocols; Theory of computation → Distributed algorithms; Security and privacy → Cryptography

**Keywords and phrases** broadcast, partial broadcast, minicast, general adversary, general network

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2020.25

**Related Version** A full version of the paper is available at <https://eprint.iacr.org/2020/1408>.

## 1 Introduction

One of the most fundamental problems in distributed computing is to achieve consistency guarantees among parties, even if some of the parties behave arbitrarily. A core primitive to achieve global consistency is broadcast. More concretely, the Byzantine broadcast problem [10] is described as follows: A designated party, called the sender, intends to distribute a value consistently among  $n$  parties such that all honest parties obtain the same value, even if the sender and/or some of the other parties behave in a malicious manner; if the sender is honest, then all honest parties agree on the sender's value.



© Chen-Da Liu-Zhang, Varun Maram, and Ueli Maurer;  
licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 25; pp. 25:1–25:16

Leibniz International Proceedings in Informatics



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

Broadcast is an important primitive that has applications in many protocols for secure multi-party computation (MPC) – as defined in [12] and [6]. It is used to implement different protocols for secure bidding, voting, collective contract signing, just to name a few. With the recent trends in research in mind, another application worth mentioning is related to cryptocurrencies, where users’ transactions are to be broadcast securely among all the nodes of the underlying blockchain network even when some of the nodes could behave arbitrarily.

## 1.1 Motivation

The seminal result of Pease et al. [10] (also [1], [2]) shows that in the standard communication model of a complete synchronous network of pairwise authenticated channels, perfectly-secure broadcast is achievable if and only if less than a third of the parties are corrupted (i.e.,  $t < n/3$ ). The fundamental reason why the bound  $t < n/3$  is tight is that a corrupted node can consistently send different messages to correct processors and make them agree on different values. To avoid this, several researchers have considered using stronger communication primitives such as partial broadcast channels, which guarantee that a message is consistent among all recipients on the channel. Hence, a natural question is to investigate a generalization of the classical broadcast problem, namely the trade-off between the strength of the communication primitives and the corruptive power from the adversary.

Most results which study such trade-offs for broadcast achievability are phrased in the so-called  $b$ -minicast model [4, 3, 9], i.e., a network which contains partial broadcast channels among any subset of parties with size at most  $b$ . But one can go beyond this *threshold* characterization of communication models (similar to the adversaries seen above modelled by a threshold  $t$ ) by considering a *general network* where the set of minicasts of size at most  $b$  among the  $n$  parties may not be complete.

To the best of our knowledge, current works on such general networks [11, 8] focus on the problem of Byzantine agreement for the concrete case of 3-minicast channels, and against a threshold adversary in the range  $n/3 \leq t < n/2$ . We continue the line of research w.r.t. general  $b$ -minicast channels. We remark that – as noted in [3] – when  $b > 3$ , perfectly secure broadcast can be realized even when there is no honest majority, in contrast to Byzantine agreement. Surprisingly, there is a lack of literature devoted to this generalization. This paper thus attempts to lay out some significant starting steps towards research in this direction.

## 1.2 Related Work

**Complete/Threshold Networks.** Many of the previous results assume a complete network of partial broadcast channels up to a certain size. Fitzi and Maurer [4] showed that assuming partial broadcast channels among every triplet of parties, global broadcast can be realized if and only if  $t < n/2$ . Considine et al. [3] generalized this result to the  $b$ -minicast model, i.e. a partial broadcast channel among any  $b$  parties, where it was shown that broadcast is achievable if and only if  $t < \frac{b-1}{b+1}n$ .

Apart from generalizing the communication primitives, one can also generalize the adversary model to *general adversary structures*. The classical problem [10] focuses on adversaries that, for a threshold  $t$ , can corrupt any subset of parties  $a$  such that  $|a| \leq t$ . This was later extended to a generalized characterization of the adversary  $\mathcal{A}$ , where it can corrupt a set of parties  $a$  such that  $a \in \mathcal{A}$  for a monotone set of subsets of the  $n$  parties [7, 5].

It was shown by Fitzi and Maurer [5] that secure broadcast can be realized from point-to-point channels if and only if there are no three sets of parties in the adversary structure that can cover the whole party set. Finally, Raykov [9] unified the previous results by studying

the feasibility of broadcast in the  $b$ -minicast model that is secure against general adversaries. Specifically, Raykov proved that broadcast is achievable from  $b$ -minicast channels against adversary structures  $\mathcal{A}$  if and only if  $\mathcal{A}$  satisfies the so-called  $(b + 1)$ -chain-free condition.

**General Networks.** Current works on general network structures with partial broadcast channels focus on the achievability of Byzantine agreement. Given that Byzantine agreement is achievable from bilateral channels if  $t < \frac{n}{3}$  and not well defined for  $t \geq \frac{n}{2}$ , they focus on the case where the network of partial broadcast channels only contains 3-minicasts (in addition to bilateral channels), and the adversary is in the range  $n/3 \leq t < n/2$ . Ravikant et al. [11] provide necessary and sufficient conditions for general 3-minicast networks to satisfy so that Byzantine agreement can be achieved while tolerating threshold adversaries in the range  $n/3 \leq t < n/2$ . In a follow-up work, Jaffe et al. [8] provide asymptotically tight bounds on the number of necessary and sufficient 3-minicast channels to construct Byzantine agreement for the same threshold adversary.

### 1.3 Contributions

We extend the results for general 3-minicast networks to general  $b$ -minicast networks and address open questions posed in both of the papers [8, 9], namely to study broadcast achievability in general communication models where only a subset of  $b$ -minicast channels may be available.

The contributions of the paper are three-fold. First, we propose a simple hierarchy of all possible adversary structures with respect to  $n$  parties, by imposing a partial order based on the  $b$ -chain terminology introduced by Raykov [9]. This allows us to analyze the feasibility of broadcast in general networks in a meaningful way. We believe this hierarchy of general adversaries could be of independent interest to the broader area of secure MPC.

Second, we present necessary conditions on general network structures for secure broadcast to be possible against general adversaries. To be precise, for each of the adversary classes in the above hierarchy, we identify types of minicast channels that are essential in *any* network in order to achieve broadcast.

Finally, we provide sufficient conditions towards achieving broadcast in general networks while tolerating general adversaries. That is, given any adversary belonging to one of the hierarchy classes, we construct a broadcast protocol for networks satisfying the sufficiency condition corresponding to that adversary. We also show that these conditions are non-trivial in the sense that they do not always require a complete set of minicast channels to begin with; w.r.t. certain weak adversaries in each class, there exist general networks with an *incomplete* set of minicasts that can still realize global broadcast using our protocol.

Our results generalize previous works in communication models assuming partial broadcast channels [4, 11, 8, 9]. We show an example in Table 1 with 6 parties  $P = \{P_1, \dots, P_6\}$ . Against a threshold adversary that can corrupt up to 3 parties, it is known that with the network structure containing all 3-minicasts,  $\mathcal{N}_3$ , broadcast is impossible, whereas with a network structure containing all 4-minicasts,  $\mathcal{N}_4$ , broadcast is possible. We depict the network  $\mathcal{N} = \mathcal{N}_4 \setminus \{P_1, P_2, P_4, P_5\}, \{P_1, P_2, P_4, P_6\}, \{P_1, P_3, P_4, P_6\}, \{P_1, P_3, P_4, P_5\}$ , for which broadcast was unknown to be impossible. For the same network structure, we show in addition that broadcast is possible with respect to the adversary structure  $\mathcal{A} = \{\{P_1, P_4, P_5\}, \{P_1, P_4, P_6\}, \{P_2, P_3, P_5\}, \{P_2, P_3, P_6\}\}$ .

■ **Table 1** In the first column, we describe network structures among 6 parties in line with Definition 3. The first two entries are related to the complete 3-minicast and 4-minicast models respectively (cf. Definition 4). We depict an incomplete 4-minicast network in the third entry where the 4-minicasts that are not available are  $\{P_1, P_2, P_4, P_5\}, \{P_1, P_2, P_4, P_6\}, \{P_1, P_3, P_4, P_6\}$  and  $\{P_1, P_3, P_4, P_5\}$ . The adversary is indicated in the second column. In the first three entries the adversary can corrupt up to  $t = 3$  parties, whereas in the last entry he can corrupt any element in  $\mathcal{A} = \{\{P_1, P_4, P_5\}, \{P_1, P_4, P_6\}, \{P_2, P_3, P_5\}, \{P_2, P_3, P_6\}\}$ . We then indicate whether broadcast can be realized securely with the corresponding network/adversary w.r.t. *any* sender.

Network	Adversary	Broadcast possible	Literature
$\mathcal{N}_3 = \{p \mid p \subseteq P \text{ and }  p  \leq 3\}$	$t = 3$	No	[3, 9]
$\mathcal{N}_4 = \{p \mid p \subseteq P \text{ and }  p  \leq 4\}$	$t = 3$	Yes	[3, 9]
	$t = 3$	No	<b>This work</b>
	$\mathcal{A}$	Yes	

### 1.4 Techniques

Here we give a higher-level overview for some of the technical ideas behind our results. As briefly mentioned above, we rely on a particular characterization of general adversary structures  $\mathcal{A}$  based on whether or not  $\mathcal{A}$  contains a so-called  $b$ -chain; if not, then  $\mathcal{A}$  is said to be  $b$ -chain-free and vice-versa. This condition was introduced in [9] and was in turn inspired by a broadcast impossibility proof of [3]. Consider a chain (or ordering) of  $b + 1$  parties, namely  $(P_1, \dots, P_{b+1})$ . Then it was shown in [3] that no protocol can realize broadcast among these  $b + 1$  parties in the complete  $b$ -minicast model when any pair of adjacent parties  $(P_i, P_{1+(i \bmod b+1)})$  ( $i = 1, \dots, b + 1$ ) can be honest while the remaining parties are corrupted by an adversary. Raykov [9] then generalized this type of corruption to a chain of party subsets, where  $(\mathcal{P}_1, \dots, \mathcal{P}_{b+1})$  is now a partition of  $n$  parties into  $b + 1$  non-empty subsets  $\mathcal{P}_i$ . He later shows that when there is such a  $(b + 1)$ -partition of  $n$  parties such that the subset of parties  $(\mathcal{P}_i \cup \mathcal{P}_{1+(i \bmod b+1)})$  ( $i = 1, \dots, b + 1$ ) can be honest while the remaining parties are corrupted by an adversary, then broadcast is impossible among  $n$  parties in the complete  $b$ -minicast model, via a straightforward reduction to the setting with  $b + 1$  parties considered in [3]. In this context, the partition is called a  $(b + 1)$ -chain and the corrupting adversary is said to contain a  $(b + 1)$ -chain. But what is surprising is that this condition is tight in the sense that, if an adversary does not contain a  $(b + 1)$ -chain, then broadcast is achievable in the  $b$ -minicast model. Namely,

► **Theorem 1** ([9, Theorem 1]). *In the complete  $b$ -minicast communication model, broadcast tolerating adversary structure  $\mathcal{A}$  is achievable if and only if  $\mathcal{A}$  is  $(b + 1)$ -chain-free.*

**Hierarchy of Adversaries.** We partition the space of all possible general adversary structures w.r.t.  $n$  parties into  $(n - 1)$  classes based on the  $b$ -chain-free condition introduced in [9]. The class  $\mathfrak{A}^{(b)}$ , for  $b \geq 3$ , is the set of adversaries that contain a  $b$ -chain but are  $(b + 1)$ -chain-free. Given a general adversary  $\mathcal{A}$ , to study the feasibility of broadcast in a general network  $\mathcal{N}$ , we



consider the unique class  $\mathcal{A}$  belongs to – say  $\mathcal{A} \in \mathfrak{A}^{(b)}$ , then because of Theorem 1, broadcast tolerating  $\mathcal{A}$  is impossible in the complete  $(b - 1)$ -minicast model, but is possible in the complete  $b$ -minicast model. This allows us to analyze the  $b$ -minicast channels in  $\mathcal{N}$  which are necessary or which suffice to achieve broadcast securely against  $\mathcal{A}$ .

**Necessary Conditions.** Given an adversary  $\mathcal{A} \in \mathfrak{A}^{(b)}$ , we identify certain types of  $b$ -minicast channels that are necessary to realize secure broadcast among parties  $P$  in any general network. We proceed to show it by starting with a complete  $b$ -minicast model, and removing  $b$ -minicast channels of the aforementioned type. Then we prove that any protocol that achieves broadcast among  $P$  in the resulting network against  $\mathcal{A}$  can be reduced to a protocol that achieves broadcast in a setting with  $b$  parties in the complete  $(b - 1)$ -minicast model against an adversary that contains a  $b$ -chain. Because the latter is deemed to be impossible by Theorem 1, we conclude that secure broadcast protocols cannot exist when such *essential*  $b$ -minicast channels are missing from a network.

**Sufficient Conditions.** Again given an adversary  $\mathcal{A} \in \mathfrak{A}^{(b)}$ , we identify a set  $S$  of  $b$ -minicast channels such that, for any general network  $\mathcal{N}$ , it suffices to have  $\mathcal{N}$  contain the minicast channels  $S$  in order to achieve broadcast secure against  $\mathcal{A}$  (assuming sufficient connectivity w.r.t.  $(b - 1)$  and lower minicast channels in  $\mathcal{N}$ ). For ease of exposition, we consider general networks  $\mathcal{N}$  with an underlying complete set of  $(b - 1)$  minicast channels. Now since  $\mathcal{A}$  is  $(b + 1)$ -chain-free, secure broadcast is possible in the complete  $b$ -minicast model according to Theorem 1. Hence the idea is to simulate the  $b$ -minicast model on the general network  $\mathcal{N}$  using its (possibly) incomplete set of  $b$ -minicast channels and the complete set of  $(b - 1)$ -minicasts underneath.

Towards finding  $S$ , we focus on its complement set  $S^c$ , namely the set of  $b$ -minicast channels which are not required to be present in  $\mathcal{N}$  to realize broadcast tolerating  $\mathcal{A}$ . If the  $b$ -minicast channels of  $S^c$  were to be missing in  $\mathcal{N}$ , it should be possible to simulate them via a local application of the feasibility result of Theorem 1, i.e., subsets of  $b$  parties that have their corresponding  $b$ -minicast channel missing could simulate partial broadcast among themselves by executing Raykov’s protocol [9] using their underlying  $(b - 1)$ -minicast channels. In that case, we have to formally argue that  $\mathcal{A}$ ’s corrupting power when restricted to these  $b$  parties is  $b$ -chain-free.

We also show that our set  $S$  for sufficiency of broadcast need not be the *trivial* complete set of  $b$ -minicast channels for all adversaries in  $\mathfrak{A}^{(b)}$ . Specifically, we identify certain weak adversaries of the class  $\mathfrak{A}^{(b)}$ , which we call  *$b$ -chain adversaries*, and prove that the set  $S^c$  is non-empty for such adversaries. Our arguments here are related to the  $b$ -chain property of [9] and are mostly combinatorial. They revolve around showing the (non-)existence of special configurations of parties placed in *bins* which are arranged in a circular fashion.

## 2 Models and Definitions

In this section, we introduce the main concepts, along with some notation, that will be used in this paper – which includes a description of the models of communication and corruption with respect to a set of parties being considered. Most of the definitions are borrowed from [9] and [3]. In this paper we consider a setting where parties do not have a public-key infrastructure (PKI) available. Note that if one assumes a PKI, it would allow messages to be signed and broadcast would be possible with arbitrary resilience.

## 2.1 Parties

Unless stated otherwise, we always consider a setting with  $n$  parties, namely  $P = \{P_1, \dots, P_n\}$ . We now describe some notions with respect to the partitions of the party set  $P$ .

► **Definition 2.** A list  $\mathcal{S} = (\mathcal{S}_0, \dots, \mathcal{S}_{k-1})$  is a  $k$ -partition of  $P$  if  $\bigcup_{i=0}^{k-1} \mathcal{S}_i = P$  and all  $\mathcal{S}_i$  and  $\mathcal{S}_j$  are pair-wise disjoint. Such partitions are said to be proper if all  $\mathcal{S}_i$  are non-empty.

In addition, we present a notation to denote the set of parties from  $P$  minus the two sets  $\mathcal{S}_i$  and  $\mathcal{S}_j$  from the  $k$ -partition  $\mathcal{S}$ :

$$\mathcal{S}_{\downarrow i,j} := P \setminus (\mathcal{S}_{i \bmod k} \cup \mathcal{S}_{j \bmod k})$$

## 2.2 Communication Network

In the classical model [10], parties are connected by a complete, synchronous network of bilateral authenticated channels. Such communication channels between any two parties  $P_i$  and  $P_j$  guarantee that only the aforementioned pair can send messages on the channel – no third party can access (or block) the channel in any way other than possibly reading the communication between  $P_i$  and  $P_j$ .

A synchronous network here means that all parties share common clock cycles. In a particular clock cycle, each party initially receives a finite (possibly empty) set of messages from the other parties, performs a finite (possibly zero) number of local computations, and finally sends a finite (possibly empty) set of messages to each other party. Additionally, it is guaranteed that the messages sent during a clock cycle arrive at the beginning of the next cycle.

We focus on a very general characterization of such communication models where, in addition to pairwise channels, there could be *partial broadcast* channels among the parties of  $P$  that allow messages to be consistently delivered to more than one recipient, i.e., channels which allow broadcast to be realized locally within certain subsets of parties.

► **Definition 3 (General network).** A general network  $\mathcal{N}$  among a set of parties  $\mathcal{P}$  is a monotone<sup>1</sup> set of subsets of  $P$ .

Given a general network  $\mathcal{N}$ , we have  $\{P_{i_1}, \dots, P_{i_k}\} \in \mathcal{N}$  if and only if there is a partial broadcast channel among  $\{P_{i_1}, \dots, P_{i_k}\}$  – based on the sizes of the subset of parties, such channels are also known as  *$k$ -minicast channels* [9].

The partial broadcast channels also provide authentication, similar in spirit to the bilateral channels, and are synchronous. Also, observe that the classical model with bilateral channels can be seen as a particular network structure  $\mathcal{N}$ , where  $\mathcal{N}$  contains nothing but all possible subsets of  $P$  with size 2. In the same way, the complete  $b$ -minicast model is a network structure which contains all partial broadcasts of size at most  $b$ .

► **Definition 4 (b-minicast model).** A complete  $b$ -minicast model is a network structure  $\mathcal{N}_b$  that contains all possible subsets of  $P$  with size at most  $b$ , i.e.,  $\mathcal{N}_b = \{p \mid p \subseteq P \text{ and } |p| \leq b\}$

Finally, we define a subclass of general networks which, roughly speaking, do not contain  $(b + 1)$  and higher minicast channels.

► **Definition 5 (General b-minicast network).** A general  $b$ -minicast network is any network structure  $\mathcal{N}$  such that  $\mathcal{N} \subseteq \mathcal{N}_b$ , where  $\mathcal{N}_b$  is the complete  $b$ -minicast model.

<sup>1</sup> If  $N \in \mathcal{N}$  and  $N' \subseteq N$  then  $N' \in \mathcal{N}$ .

## 2.3 Adversary and Security

We assume the existence of a central adversary that corrupts a subset of parties and makes them deviate from a protocol in an arbitrary way. Such corrupted parties are often called *Byzantine*, and the parties that are not corrupted will be referred to as *honest* or *correct*. Our protocols are *perfectly secure*, which guarantees that a protocol never fails (zero probability of error) even under computationally unbounded adversaries. On the other hand, our impossibility results hold even against a bounded adversary.

In the paper, we mainly consider a general adversary structure  $\mathcal{A}$  [7], which specifies the possible subsets of parties that the adversary can corrupt. We require that  $\mathcal{A}$  be monotone, i.e.,  $\forall a, a' (a \in \mathcal{A} \text{ and } (a' \subseteq a) \implies a' \in \mathcal{A}$ .

► **Definition 6** (General adversary). *A general adversary  $\mathcal{A}$  among a set of parties  $P$  is a monotone set of subsets of  $P$ .*

In the literature, a specific type of adversary is widely discussed, namely a *threshold* adversary, that is characterized by the maximum number of parties  $t$  which can be corrupted; the adversary structure takes the following form  $\mathcal{A}_t = \{a \mid a \subseteq P \text{ and } |a| \leq t\}$ . A typical non-threshold adversary structure is the so-called  $\mathcal{Q}^{(k)}$  adversary. The adversary structure  $\mathcal{A}$  is said to satisfy  $\mathcal{Q}^{(k)}$  if the union of no  $k$  sets in  $\mathcal{A}$  equals the party set  $P$ . In this paper, we are more interested in the  $k$ -chain condition, which was introduced in [9]. In the full version we briefly discuss the relation between the  $k$ -chain condition and the  $\mathcal{Q}^{(k)}$  condition.

► **Definition 7.** *An adversary structure  $\mathcal{A}$  is said to contain a  $k$ -chain if there exists a proper  $k$ -partition  $\mathcal{S} = (\mathcal{S}_0, \dots, \mathcal{S}_{k-1})$  of the party set  $P$  such that  $\forall i \in [0, k-1] \mathcal{S}_{\downarrow i, i+1} \in \mathcal{A}$ . An adversary structure is  $k$ -chain-free if it does not have a  $k$ -chain.*

## 2.4 Broadcast

In broadcast, a designated party (known as the sender) wants to distribute its input value, i.e., a message, among  $n$  parties such that all honest parties receive the same message.

► **Definition 8** (Broadcast). *A protocol among the party set  $P$  where some specific party  $P_s \in P$  (the sender) holds an input  $v \in \mathcal{D}$  and each party  $P_i \in P$  outputs a value  $y_i \in \mathcal{D}$  achieves broadcast if the following holds:*

Validity: *If the sender  $P_s$  is honest, then every honest party  $P_i \in P$  outputs the sender's value, i.e.,  $y_i = v$ .*

Consistency: *All honest parties in  $P$  output the same value.*

## 3 Hierarchy of Adversary Structures

Let us consider any general adversary  $\mathcal{A}$  with respect to the parties  $P$ . Ignoring the two extreme cases where the adversary is either too weak that broadcast is achievable by only using bilateral channels (i.e.,  $\mathcal{A}$  is 3-chain-free, see Theorem 1), or when the adversary is too strong that secure broadcast is not possible among the  $n$  parties unless we assume a global broadcast primitive in the first place (i.e.,  $\mathcal{A}$  contains an  $n$ -chain), we note that there must exist  $b \in [3, n-1]$  such that  $\mathcal{A}$  contains a  $b$ -chain and is  $(b+1)$ -chain-free.

This observation gives us a way to define a simple partial order over the complete space of general adversarial structures with respect to the party set  $P$ , which is highly relevant to the problem of achieving secure broadcast in general communication models. Define the

weakest and strongest class of adversaries respectively as,

$$\mathfrak{A}^{(0)} = \{\mathcal{A} \subseteq 2^P \mid \mathcal{A} \text{ is 3-chain-free}\}$$

$$\mathfrak{A}^{(n)} = \{\mathcal{A} \subseteq 2^P \mid \mathcal{A} \text{ contains an } n\text{-chain}\}$$

The subsequent classes of adversary structures in-between are defined as:  $\forall b \in [3, n-1]$ ,

$$\mathfrak{A}^{(b)} = \{\mathcal{A} \subseteq 2^P \mid \mathcal{A} \text{ contains a } b\text{-chain and is } (b+1)\text{-chain-free}\}$$

The classes of adversaries arranged in an increasing order of strength would then be  $\mathfrak{A}^{(0)} \leq \mathfrak{A}^{(3)} \leq \mathfrak{A}^{(4)} \leq \dots \leq \mathfrak{A}^{(n)}$ . Note that this forms indeed a *partition* over all possible adversary structures, since for  $b > 3$ , any adversary structure in  $\mathfrak{A}^{(b)}$  also contains an implicit  $(b-1)$ -chain (and lower). This can be seen from the definition of a  $b$ -chain-free adversary structure. More concretely, if an adversary structure  $\mathcal{A}$  is  $b$ -chain-free, then it is also  $(b+1)$ -chain-free. This is because if it contains a  $(b+1)$ -chain  $(S_0, \dots, S_b)$ , then  $(S_0, \dots, S_{b-2}, S_{b-1} \cup S_b)$  is a valid  $b$ -chain, since  $\mathcal{A}$  is monotone. In fact, we also show in Subsection 5.1 that the partition is a proper partition, i.e., each set  $\mathfrak{A}^{(b)}$  is non-empty.

Given this partition, one can consider a partial order over adversary structures as follows: we define the order relation  $\prec$  such that for any two general adversaries  $\mathcal{A}_p \in \mathfrak{A}^{(k)}$  and  $\mathcal{A}_q \in \mathfrak{A}^{(k')}$ :  $\mathcal{A}_p \prec \mathcal{A}_q \iff (k < k') \vee (k = k' \wedge \mathcal{A}_p = \mathcal{A}_q)$ .

As mentioned in Subsection 1.4, given a general adversary  $\mathcal{A}$ , there is a unique class  $\mathfrak{A}^{(b)}$  that  $\mathcal{A}$  belongs to. We know that broadcast tolerating  $\mathcal{A}$  is impossible in the complete  $(b-1)$ -minicast model, but is possible in the complete  $b$ -minicast model because of Theorem 1. We can then analyze the  $b$ -minicast channels which are necessary or which suffice to achieve broadcast in a general network securely against  $\mathcal{A}$ .

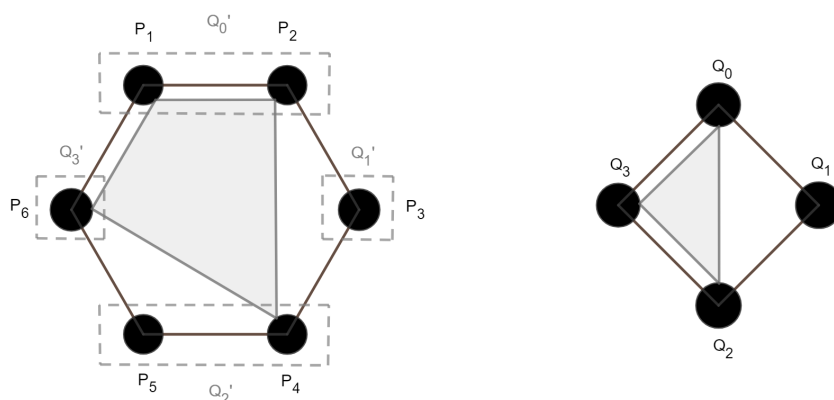
## 4 Necessary Conditions

In this section, we provide necessary conditions on general communication networks for broadcast to be possible while tolerating general adversaries. Specifically, given an adversary structure  $\mathcal{A} \in \mathfrak{A}^{(b)}$ , we identify some types of minicast channels that have to be present in any network  $\mathcal{N}$  in order to achieve secure broadcast.

Let  $\mathcal{P} = (\mathcal{P}_0, \dots, \mathcal{P}_{b-1})$  be any  $b$ -chain that is present in  $\mathcal{A}$ . We then characterize  $b$ -minicasts that are essential for broadcast against  $\mathcal{A}$  in a general communication network, namely of the form  $\{p_0, \dots, p_{b-1}\}$  where  $p_0 \in \mathcal{P}_0, \dots, p_{b-1} \in \mathcal{P}_{b-1}$ . At a very high level, the proof uses any protocol that achieves broadcast in an incomplete  $b$ -minicast network – with the aforementioned essential  $b$ -minicasts missing – against  $\mathcal{A} \in \mathfrak{A}^{(b)}$  in order to construct a broadcast protocol among  $b$  parties in the complete  $(b-1)$ -minicast model against an adversary containing a  $b$ -chain. Since the latter is known to be impossible by Theorem 1, the former is also impossible. We depict in Figure 1 a concrete example of the reduction for the case of  $b = 4$ .

► **Lemma 9.** *Secure broadcast on a general network  $\mathcal{N}$  tolerating a general adversary  $\mathcal{A} \in \mathfrak{A}^{(b)}$  is possible for some sender only if: for every  $b$ -chain in  $\mathcal{A}$ , namely  $\mathcal{P} = (\mathcal{P}_0, \dots, \mathcal{P}_{b-1})$ , there is a  $b$ -minicast channel in  $\mathcal{N}$  that has non-empty intersection with the sets  $\mathcal{P}_0, \dots, \mathcal{P}_{b-1}$ .*

**Proof.** Consider any  $b$ -chain in  $\mathcal{A}$ ,  $\mathcal{P} = (\mathcal{P}_0, \dots, \mathcal{P}_{b-1})$ . From a complete network structure  $\mathcal{N}_{comp} = 2^P$ , we remove  $b$ -minicast channels of the form  $\{p_0, \dots, p_{b-1}\}$  where  $p_0 \in \mathcal{P}_0, \dots, p_{b-1} \in \mathcal{P}_{b-1}$ . Because the set of partial broadcast channels is monotone (cf. Definition 3), all higher  $b'$ -minicasts (with  $b' > b$ ) that *contain* a  $b$ -minicast of the removed type will be missing implicitly as well (more formally, if  $S = \{p_0, \dots, p_{b-1}\}$ , then we are talking about  $b'$ -minicasts shared by the set  $S'$  of  $b'$  parties such that  $S' \supseteq S$ ).



■ **Figure 1** On the left, there is an adversary containing a 4-chain  $(Q'_0, Q'_1, Q'_2, Q'_3)$  among 6 parties in the incomplete 4-minicast network missing all 4-minicasts that have non-empty intersection with the sets  $Q'_0, \dots, Q'_3$ . On the right, there is an adversary with a 4-chain  $(Q_0, Q_1, Q_2, Q_3)$  among 4 parties in the complete 3-minicast network. Now our reduction lets each party  $Q_i$  emulate the set of parties  $Q'_i$ . Then, any 4-minicast present on the left can be emulated by a 3-minicast on the right. For example, the 4-minicast depicted in light-gray on the left can be emulated using the 3-minicast depicted on the right.

Assume there exists a broadcast protocol  $\pi$  for some sender  $P_s$  from  $P$  tolerating the adversary  $\mathcal{A}$ . Using  $\pi$ , we now construct a protocol  $\pi'$  for achieving broadcast among  $b$  parties  $\{Q_0, \dots, Q_{b-1}\}$  in the complete  $(b-1)$ -minicast communication model under an adversary containing a  $b$ -chain, i.e., any pair  $(Q_i, Q_{(i+1) \bmod b})$  can be honest while the adversary corrupts the rest of the parties. Since from Theorem 1 such protocol  $\pi'$  cannot exist, we arrive at a contradiction.

To construct protocol  $\pi'$  from  $\pi$ , the protocol  $\pi'$  lets each  $Q_i$  simulate the set  $\mathcal{P}_i$ . The key thing to note is that all partial broadcast channels that will be used among  $P$  in the execution of  $\pi$  can be simulated by these  $b$  parties in the  $(b-1)$ -minicast model. On the other hand, the simulation of channels that were removed in the first place would have required a (non-existent) global broadcast channel  $\{Q_0, \dots, Q_{b-1}\}$ . Also all possible corruptions by an adversary containing a  $b$ -chain, among  $\{Q_0, \dots, Q_{b-1}\}$  is already covered by the adversary structure  $\mathcal{A}$  with respect to  $\pi$ , because  $\mathcal{P}$  is a  $b$ -chain contained in  $\mathcal{A}$ , and if  $Q_i$  is corrupted, all the parties simulated by it  $\mathcal{P}_i$  can behave arbitrarily. Since protocol  $\pi$  is assumed to be secure against  $\mathcal{A}$ ,  $\pi'$  allows the party that simulates  $P_s$  to broadcast securely, thereby arriving at the contradiction. ◀

In fact, we can extend the above simulation argument to have a similar characterization of essential partial broadcast channels for every lower  $k$ -chain ( $k < b$ )  $\mathcal{P} = (\mathcal{P}_0, \dots, \mathcal{P}_{k-1})$  contained in  $\mathcal{A}$ , i.e., minicasts of the type  $\{p_0, \dots, p_{k-1}\}$  where  $p_0 \in \mathcal{P}_0, \dots, p_{k-1} \in \mathcal{P}_{k-1}$ . This leads us to our necessary condition for secure broadcast in general network structures.

▶ **Theorem 10.** *Secure broadcast on a general network  $\mathcal{N}$  and against a general adversary  $\mathcal{A} \in \mathfrak{A}^{(b)}$  is possible for some sender only if:  $\forall k \in [3, b]$ , if  $\mathcal{P} = (\mathcal{P}_0, \dots, \mathcal{P}_{k-1})$  is a  $k$ -chain in  $\mathcal{A}$ , then there must be a  $k$ -minicast channel in  $\mathcal{N}$  that has non-empty intersection with each of the sets  $\mathcal{P}_0, \dots, \mathcal{P}_{k-1}$ .*

**Sufficient Number of  $b$ -Minicasts.** Jaffe et al. [8] present asymptotically tight bounds on the number of 3-minicast channels that are necessary and sufficient to achieve Byzantine agreement in general 3-minicast networks against threshold adversaries in the range  $n/3 \leq$

$t < n/2$ . One of their main results is giving a bound for the quantity  $U_n(t)$  which is the minimum  $m$  such that *any* general 3-minicast network  $\mathcal{N}$  with  $m$  3-minicast channels achieves Byzantine agreement while tolerating at most  $t$  corrupted parties – the underlying graph (2-minicast network) of  $\mathcal{N}$  is assumed to be sufficiently connected, e.g., via a complete set of bilateral channels among the  $n$  parties.

An open question in [8] was scaling its definitions and results to general  $b$ -minicast networks, for  $b > 3$ . We provide some steps towards answering it by first generalizing the definition of  $U_n(\cdot)$  in a higher  $b$ -minicast setting. Similar to [8], there we consider general  $b$ -minicast networks  $\mathcal{N}$  that have a complete set of  $(b-1)$ -minicast channels underneath.

► **Definition 11.** *Let  $U_n^b(\mathcal{A})$  denote the minimum number  $m$  such that any general network structure  $\mathcal{N}$  with  $m$   $b$ -minicast channels, and satisfying  $\mathcal{N}_{b-1} \subseteq \mathcal{N} \subseteq \mathcal{N}_b$ , achieves global broadcast for some sender among the  $n$  parties while tolerating the general adversary  $\mathcal{A}$ .*

We now present a lower bound on  $U_n^b(\mathcal{A})$ , for  $b \geq 3$  and any general adversary  $\mathcal{A} \in \mathfrak{A}^{(b)}$ . Note that this is similar in spirit to the analysis in [8] because the threshold adversaries  $n/3 \leq t < n/2$  belong to the class  $\mathfrak{A}^{(3)}$  (see the full version). Towards our bound, we consider another quantity which is like a complement to  $U_n^b(\mathcal{A})$ , namely  $u_n^b(\mathcal{A})$  which is defined as the minimum number of  $b$ -minicast channels that can be removed from a complete  $b$ -minicast model to ensure broadcast cannot be realized securely w.r.t. any sender among the  $n$  parties against  $\mathcal{A}$ . It is then not hard to see that  $U_n^b(\mathcal{A}) = \binom{n}{b} - u_n^b(\mathcal{A}) + 1$ .

So we essentially provide an upper bound on  $u_n^b(\mathcal{A})$  by removing  $b$ -minicast channels of the type described in Lemma 9 so as to violate our necessary condition for secure broadcast in the resulting incomplete  $b$ -minicast network. We do this by finding a  $b$ -chain  $\mathcal{P} = (\mathcal{P}_0, \dots, \mathcal{P}_{b-1})$  in  $\mathcal{A} \in \mathfrak{A}^{(b)}$  which minimizes, over all  $b$ -chains present in  $\mathcal{A}$ , the size of the corresponding set of essential  $b$ -minicast channels  $\{\{p_0, \dots, p_{b-1}\} \in \mathcal{N}_b \mid p_0 \in \mathcal{P}_0, \dots, p_{b-1} \in \mathcal{P}_{b-1}\}$  (here  $\mathcal{N}_b$  is the complete  $b$ -minicast model as denoted in Definition 4). To be precise, we attempt to solve the following optimization problem, over all proper  $b$ -partitions  $\mathcal{P} = (\mathcal{P}_0, \dots, \mathcal{P}_{b-1})$  of the party set  $P$  w.r.t. the adversary structure  $\mathcal{A}$ :

$$\begin{aligned} & \underset{\mathcal{P}}{\text{minimize}} && |\mathcal{P}_0| \times \dots \times |\mathcal{P}_{b-1}| \\ & \text{subject to} && \mathcal{P}_{\downarrow i, i+1} \in \mathcal{A}, \quad i = 0, \dots, b-1. \end{aligned}$$

If  $\pi_{\mathcal{A}}$  is a solution to the above problem, we have  $u_n^b(\mathcal{A}) \leq \pi_{\mathcal{A}}$  which results in the following lower bound on  $U_n^b(\mathcal{A})$ .

$$U_n^b(\mathcal{A}) \geq \binom{n}{b} - \pi_{\mathcal{A}} + 1$$

Note that the above results hold for every adversary in the class  $\mathfrak{A}^{(b)}$  – in particular, the threshold adversaries in the range  $\frac{b-2}{b}n \leq t < \frac{b-1}{b+1}n$  [3]. In the full version we explore in detail this relation.

## 5 Sufficient Conditions

In this section, we present some conditions on general networks which are sufficient to achieve global broadcast secure against general adversaries. To be specific, given an adversary structure  $\mathcal{A} \in \mathfrak{A}^{(b)}$  and a network  $\mathcal{N}$  satisfying the aforementioned sufficiency condition, we construct a protocol that realizes Byzantine broadcast for any sender in  $P$ .

The main idea is to simulate Raykov’s protocol [9] – that considers complete  $b$ -minicast models and  $(b+1)$ -chain-free adversaries – on the general network  $\mathcal{N}$ . And we do this by *patching* any  $b$ -minicast channel that might be missing in  $\mathcal{N}$  with local executions of Raykov’s

protocol among subsets of  $b$  parties. To make this intuition more rigorous, we first define an operator on the general adversary  $\mathcal{A}$  that, roughly speaking, *projects* its corruption power onto such subsets of parties.

► **Definition 12.** *Let  $\mathcal{A}$  be any general adversary over a party set  $P$ . For a given subset of parties  $S \subseteq P$ , we define the adversary structure  $\mathcal{A}[S]$  to be the projection of  $\mathcal{A}$  onto  $S$ , where  $\mathcal{A}[S] = \{a \cap S \mid a \in \mathcal{A}\}$ .*

We make a couple of observations about the projected adversary  $\mathcal{A}[S]$ . First,  $\mathcal{A}[S] \subseteq \mathcal{A}$ : Given any  $a \in \mathcal{A}[S]$ , we have  $a = a' \cap S$  for a corresponding  $a' \in \mathcal{A}$ . But since  $a \subseteq a'$  and  $\mathcal{A}$  is monotone, we have  $a \in \mathcal{A}$ . Second,  $\mathcal{A}[S]$  is a well-defined general adversary among the set of parties  $S$  in accordance with Definition 6, i.e.,  $\mathcal{A}[S]$  is a monotone set of subsets of  $S$ . It is clear that  $\mathcal{A}[S]$  is a subset of  $2^S$ , because  $\forall a \in \mathcal{A}[S]$ , we have  $a \subseteq S$ . To show monotonicity, let  $a \in \mathcal{A}[S]$  and  $a' \subseteq a$ . Then because  $\mathcal{A}[S] \subseteq \mathcal{A}$  and  $\mathcal{A}$  is monotone, we have  $a' \in \mathcal{A}$ . As  $a' \subseteq a \subseteq S$ , we have  $a' = a' \cap S$ , which implies that  $a' \in \mathcal{A}[S]$ .

Coming to our protocol idea, consider a subset  $S$  of  $b$  parties. If  $\mathcal{A}[S]$  is  $b$ -chain-free, then the network  $\mathcal{N}$  can afford to have the  $b$ -minicast channel among  $S$  to be missing, since such a minicast channel can be locally simulated by the parties  $S$  via executing Raykov's broadcast protocol [9] on their underlying  $(b-1)$ -minicast network. It is then not hard to see that this line of reasoning extends recursively w.r.t.  $(b-1)$  and lower minicast channels. That is, now a  $(b-1)$ -minicast channel among  $S$ , say the one shared by a subset of parties  $S' \subset S$  with  $|S'| = b-1$ , is not required by our protocol if the projected adversary  $\mathcal{A}[S']$  is  $(b-1)$ -chain-free. For this recursion to terminate, we assume a complete set of bilateral channels in the network  $\mathcal{N}$ . Thus, we arrive at the following sufficient condition for secure broadcast in general networks.

► **Theorem 13.** *Secure broadcast on a general network  $\mathcal{N}$  and against a general adversary  $\mathcal{A} \in \mathfrak{A}^{(b)}$  is possible for any sender if:*

- $\mathcal{N}$  contains a complete set of bilateral channels, i.e.  $\mathcal{N}_2 \subseteq \mathcal{N}$ .
- For each subset of parties  $S$  of size  $k$ , where  $3 \leq k \leq b$ : if  $\mathcal{A}[S]$  contains a  $k$ -chain, there is a  $k$ -minicast channel in  $\mathcal{N}$  among  $S$ .

## 5.1 Chain Adversaries

At first glance, our sufficient condition above may seem to require the general network to have a complete set of  $b$ -minicast channels in the first place. We show that this is not the case for a certain class of adversaries – *chain adversaries* – which we define as follows.

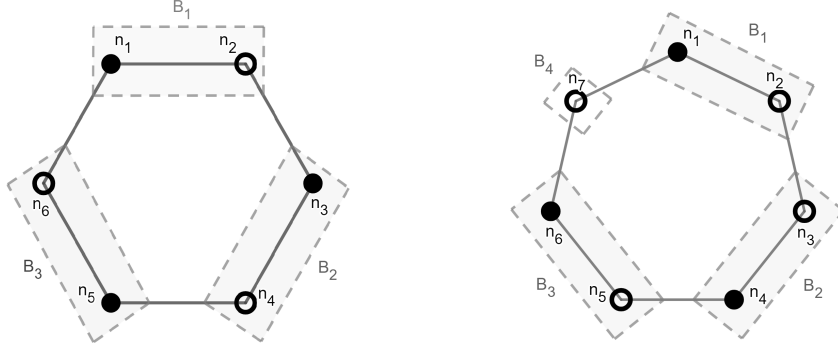
► **Definition 14.** *Let the list  $\mathcal{P} = (\mathcal{P}_0, \dots, \mathcal{P}_{b-1})$  be any proper  $b$ -partition of  $P$ . Then we define a corresponding  $b$ -chain adversary:*

$$\mathcal{A}^{\mathcal{P}} = \{a \mid \exists i \in [0, b-1] \text{ such that } a \subseteq \mathcal{P}_{\downarrow i, i+1}\}$$

It is clear that  $\mathcal{A}^{\mathcal{P}}$  contains a  $b$ -chain, since we have a proper  $b$ -partition  $\mathcal{P}$  such that  $\forall i \in [0, b-1] \mathcal{P}_{\downarrow i, i+1} \in \mathcal{A}^{\mathcal{P}}$ . From Theorem 1, we note that these  $b$ -chain adversaries are precisely the *minimal* adversary structures under which broadcast is impossible in the  $(b-1)$ -minicast model. So one would expect them to be weaker (i.e., broadcast to be possible) when assuming a  $b$ -minicast model. We show that this is indeed the case by formally proving that  $b$ -chain adversaries are  $(b+1)$ -chain-free, i.e.,  $\mathcal{A}^{\mathcal{P}} \in \mathfrak{A}^{(b)}$ .

Before proceeding with the technical details, we describe a setting that will be helpful to understand our proofs of the following results. When arguing about (ordered) lists such as  $\mathcal{S} = (\mathcal{S}_0, \dots, \mathcal{S}_{b-1})$  that are  $b$ -chains w.r.t. general adversaries, it helps to think of their





■ **Figure 2** Examples for  $b = 6$  and  $b = 7$ .

partition sets (i.e., the  $\mathcal{S}_i$ 's) as being arranged in a circular fashion, among which there is a collection of  $\lceil b/2 \rceil$  bins. More precisely,  $\forall k \in [1, \lceil b/2 \rceil - 1]$ , define  $B_k = (\mathcal{S}_{2k-2} \cup \mathcal{S}_{2k-1})$ , and depending on whether  $b$  is even or odd, the last bin  $B_{\lceil b/2 \rceil}$  is either  $(\mathcal{S}_{b-2} \cup \mathcal{S}_{b-1})$  or  $\mathcal{S}_{b-1}$  respectively. Figure 2 describes the setting of Lemma 15 where each  $\mathcal{S}_i = \{n_{i+1}\}$  is a singleton set. It depicts some simpler examples of  $b = 6$  and  $b = 7$ . For the diagram on the left, we consider a subset  $S_N = \{n_1, n_3, n_5\}$ , and on the right, we consider  $S_N = \{n_1, n_4, n_6\}$ . The  $n_i$ 's are represented as vertices of a regular  $b$ -gon; a bold vertex means that the corresponding  $n_i$  is included in  $S_N$ , and a hollow vertex implies that  $n_i \notin S_N$ . A bin  $B_j$  is lightly shaded if it is covered by  $S_N$ , i.e., either  $n_{2j-1} \in S_N$  or  $n_{2j} \in S_N$ , or more generally,  $B_j \cap S_N \neq \emptyset$ . In the example for  $b = 7$  detailed in Figure 2,  $B_2$  is covered as  $n_4 \in S_N$  whereas  $B_4$  is not because  $n_7 \notin S_N$ .

Now towards proving that  $b$ -chain adversaries are indeed  $(b+1)$ -chain-free, we start with the following combinatorial lemma that considers a simpler setting with  $b$  entities (or parties), namely  $\{n_1, \dots, n_b\}$ . The proof is in the full version and uses the above methodology with bins.

► **Lemma 15.** Let  $N = \{n_1, \dots, n_b\}$  and the list  $\mathcal{S} = (\{n_1\}, \dots, \{n_b\})$  be a  $b$ -partition of  $N$ . Define  $\mathcal{S}_{\downarrow i, i+1} = N \setminus \{n_i, n_{1+(i \bmod b)}\}$  for  $i = 1, \dots, b$ .

1. For all subsets  $S_N \subseteq N$  with  $|S_N| < \lceil b/2 \rceil \exists i \in [1, b]$  such that  $S_N \subseteq \mathcal{S}_{\downarrow i, i+1}$ .
2. There exist subsets  $S_N \subseteq N$  with  $|S_N| = \lceil b/2 \rceil$  such that  $\forall i \in [1, b] S_N \not\subseteq \mathcal{S}_{\downarrow i, i+1}$ .

► **Lemma 16.** Let  $\mathcal{A}^P$  be any  $b$ -chain adversary. Then  $\mathcal{A}^P \in \mathfrak{A}^{(b)}$ .

**Proof.** As discussed before,  $\mathcal{A}^P$  already contains a  $b$ -chain, namely  $\mathcal{P}$ . Now we show that the adversary structure is also  $(b+1)$ -chain-free, thereby completing the proof. Towards a contradiction, assume  $\mathcal{A}^P$  contains a  $(b+1)$ -chain, namely  $\mathcal{S} = (\mathcal{S}_0, \dots, \mathcal{S}_b)$ , where  $\forall i \in [0, b] \mathcal{S}_{\downarrow i, i+1} \in \mathcal{A}^P$ . Consider a subset of  $\lceil b/2 \rceil$  alternating list elements from  $\mathcal{P}$ , namely  $S_{\mathcal{P}} = \{\mathcal{P}_0, \mathcal{P}_2, \dots, \mathcal{P}_{2(\lceil b/2 \rceil - 1)}\}$ . From Lemma 15.2 w.r.t.  $\mathcal{P}$ , it is not hard to see that if we pick a single party from each partition set in  $S_{\mathcal{P}}$  and construct a subset of parties  $S_p = \{p_0, p_2, \dots, p_{2(\lceil b/2 \rceil - 1)}\}$ , where  $p_0 \in \mathcal{P}_0, p_2 \in \mathcal{P}_2, \dots$ , then  $\forall i \in [0, b-1] S_p \not\subseteq \mathcal{P}_{\downarrow i, i+1}$  (this can also be seen by noting that for any  $i \in [0, b-1] S_p \cap (\mathcal{P}_i \cup \mathcal{P}_{(i+1) \bmod b}) \neq \emptyset$ ). Thus because of the way  $\mathcal{A}^P$  is defined,  $S_p \notin \mathcal{A}^P$ .

Now coming to the  $(b+1)$ -chain  $\mathcal{S}$ , no matter where we assign the parties of  $S_p$  among the partition sets of  $\mathcal{S}$ , we can only consider at most  $\lceil b/2 \rceil$  of those sets. For even  $b$ , as  $\lceil \frac{b}{2} \rceil < \lceil \frac{b+1}{2} \rceil$ , using Lemma 15.1 with respect to  $\mathcal{S}$ , we note that  $\exists i \in [0, b]$  such that  $S_p \subseteq \mathcal{S}_{\downarrow i, i+1}$ , and hence,  $S_p \in \mathcal{A}^P$ , which is a contradiction.

When  $b$  is odd, we have  $\lceil \frac{b}{2} \rceil = \lceil \frac{b+1}{2} \rceil$ . Using the *bins* methodology described above, we observe that each of the  $\lceil \frac{b}{2} \rceil$  *disjoint* bins in  $\mathcal{S}$  must be covered by the  $\lceil \frac{b}{2} \rceil$  parties of  $S_p$  such that the assigned partition sets of the  $(b+1)$ -chain are alternating (the set  $\mathcal{S}_i$  is said to be *assigned* w.r.t. parties in  $S_p$  if and only if  $S_p \cap \mathcal{S}_i \neq \emptyset$ ), so as to avoid the contradiction  $S_p \in \mathcal{A}^P$ . Note that for any  $i \in [0, b]$ ,  $S_p \subseteq \mathcal{S}_{\downarrow i, i+1} \iff S_p \cap (\mathcal{S}_i \cup \mathcal{S}_{(i+1) \bmod b+1}) = \emptyset$ . Thus the contradiction occurs whenever there is a pair of consecutive *unassigned* partition sets  $(\mathcal{S}_i, \mathcal{S}_{(i+1) \bmod b+1})$ . Now without loss of generality, let the assigned sets be  $\{\mathcal{S}_0, \mathcal{S}_2, \dots, \mathcal{S}_{b-1}\}$  with respect to  $S_p$ . More formally, let  $\pi$  be a bijection from the set  $\{0, 2, \dots, b-1\}$  onto itself such that  $p_0 \in \mathcal{S}_{\pi(0)}$ ,  $p_2 \in \mathcal{S}_{\pi(2)}$ ,  $\dots$ ,  $p_{b-1} \in \mathcal{S}_{\pi(b-1)}$ . Now consider a subset of parties wherein we replace  $p_{b-1}$  in  $S_p$  with another party from  $\mathcal{P}_{b-1}$ , i.e.,  $S'_p = \{p_0, p_2, \dots, p'_{b-1}\}$  where  $p'_{b-1} \in \mathcal{P}_{b-1}$ . Again it is not hard to see from above that  $S'_p \notin \mathcal{A}^P$ . As the assignment of the parties  $p_0, p_2, \dots, p_{b-3}$  is already fixed, to preserve the cyclic arrangement of the partition sets so as to avoid a contradiction with respect to  $S'_p$ , we have  $p'_{b-1} \in \mathcal{S}_{\pi(b-1)}$ . Repeating this procedure with all parties of  $\mathcal{P}_{b-1}$ , and moving over to other  $\mathcal{P}_i$ 's, we essentially get that  $\mathcal{P}_0 \subseteq \mathcal{S}_{\pi(0)}$ ,  $\mathcal{P}_2 \subseteq \mathcal{S}_{\pi(2)}$ ,  $\dots$ ,  $\mathcal{P}_{b-1} \subseteq \mathcal{S}_{\pi(b-1)}$ . Now consider another subset of  $\lceil b/2 \rceil$  list elements from  $\mathcal{P}$ , i.e.,  $\overline{\mathcal{S}}_{\mathcal{P}} = \{\mathcal{P}_1, \mathcal{P}_3, \dots, \mathcal{P}_{b-2}, \mathcal{P}_0\}$ . Looking at the subset of parties  $\overline{\mathcal{S}}_p = \{p_1, p_3, \dots, p_{b-2}, p_0\}$ , where  $p_1 \in \mathcal{P}_1$ ,  $p_3 \in \mathcal{P}_3$ ,  $\dots$ , Lemma 15.2 again shows that  $\forall i \in [0, b-1]$   $\overline{\mathcal{S}}_p \not\subseteq \mathcal{P}_{\downarrow i, i+1}$ , and thus  $\overline{\mathcal{S}}_p \notin \mathcal{A}^P$ . As the assignment of  $p_0$  to  $\mathcal{S}_{\pi(0)}$  in the  $(b+1)$ -chain is already fixed, we observe that the assigned partition sets with respect to  $\overline{\mathcal{S}}_p$  again has to be  $\{\mathcal{S}_0, \mathcal{S}_2, \dots, \mathcal{S}_{b-1}\}$  in order to avoid contradictions. Thus, define a bijection  $\overline{\pi}$  from the set  $\{1, 3, \dots, b-2, 0\}$  onto  $\{0, 2, \dots, b-1\}$  such that  $p_1 \in \mathcal{S}_{\overline{\pi}(1)}$ ,  $p_3 \in \mathcal{S}_{\overline{\pi}(3)}$ ,  $p_{b-2} \in \mathcal{S}_{\overline{\pi}(b-2)}$ , and  $\overline{\pi}(0) = \pi(0)$ . Again we have  $\mathcal{P}_1 \subseteq \mathcal{S}_{\overline{\pi}(1)}$ ,  $\mathcal{P}_3 \subseteq \mathcal{S}_{\overline{\pi}(3)}$ ,  $\dots$ ,  $\mathcal{P}_{b-2} \subseteq \mathcal{S}_{\overline{\pi}(b-2)}$ . We see that all parties of  $P$  are distributed among the even indexed partition sets of  $\mathcal{S}$ . In particular, it means that  $\mathcal{S}_1 = \emptyset$ , thereby implying that the  $(b+1)$ -chain  $\mathcal{S}$  is not a proper partition, which is again a contradiction.

Finally, we have that  $\mathcal{A}^P$  is indeed  $(b+1)$ -chain-free, and thus belongs to the class  $\mathfrak{A}^{(b)}$ .  $\blacktriangleleft$

So w.r.t. the hierarchy of general adversary structures described in Section 3, we can view these  $b$ -chain adversaries to be the weakest within the class  $\mathfrak{A}^{(b)}$ . Now against these chain adversaries, we achieve Byzantine broadcast in general networks  $\mathcal{N}$  with missing  $b$ -minicast channels using Theorem 13. Specifically, we identify certain  $b$ -minicast channels which are not required (and thus can be missing in  $\mathcal{N}$ ) to achieve broadcast against  $b$ -chain adversaries.

► **Lemma 17.** *For  $3 < b < n$ : given any  $b$ -chain adversary  $\mathcal{A}^P \in \mathfrak{A}^{(b)}$ , there exist subsets of parties  $S$  of size  $b$  such that the projected adversary structure  $\mathcal{A}^P[S]$  is  $b$ -chain free.*

**Proof.** Towards the proof, we construct such a subset  $S \subseteq P$  with  $|S| = b$ . Let  $\mathcal{P} = (\mathcal{P}_0, \mathcal{P}_1, \dots, \mathcal{P}_{b-1})$  be the  $b$ -chain corresponding to the adversary  $\mathcal{A}^P$ . Since  $b < n$ , at least one  $\mathcal{P}_i$  contains more than one party; without loss of generality, let  $\mathcal{P}_0$  be such a set. Define the subset  $S = \{p_0, p'_0, p_1, \dots, p_{b-2}\}$  where  $\{p_0, p'_0\} \subseteq \mathcal{P}_0$ ,  $p_1 \in \mathcal{P}_1, \dots, p_{b-2} \in \mathcal{P}_{b-2}$ . We show that the projected adversary  $\mathcal{A}^P[S]$  does not contain a  $b$ -chain.

When  $b$  is even, consider the following subsets of size  $b/2$ :  $S_0 = \{p_0, p_2, \dots, p_{b-2}\}$  and  $S'_0 = \{p'_0, p_2, \dots, p_{b-2}\}$ . From Lemma 15.2, we have that  $\forall i \in [0, b-1]$   $S_0, S'_0 \not\subseteq \mathcal{P}_{i, i+1}$ , which in turn means that  $S_0, S'_0 \notin \mathcal{A}^P$ . Because we have  $\mathcal{A}^P[S] \subseteq \mathcal{A}^P$ , it is then clear that  $S_0, S'_0 \notin \mathcal{A}^P[S]$ . Now towards a contradiction, assume  $\mathcal{A}^P[S]$  contains a  $b$ -chain. Describing such a  $b$ -chain using  $b/2$  *bins*, as in the proof of Lemma 16, we observe that each element of  $S_0$  has to be put in a unique bin in an alternative fashion to ensure that  $S_0 \notin \mathcal{A}^P[S]$  – i.e., every pair of consecutive elements in the  $b$ -chain must have a non-empty intersection with

$S_0$ . Coming to  $S'_0$ , we have to place  $p'_0$  in the same bin as  $p_0$  so that every bin is covered by  $S'_0$ . But this would disrupt the alternative arrangement among  $S'_0$  which would lead to a pair of consecutive spots on the  $b$ -chain that are not occupied by  $S'_0$  resulting in  $S'_0 \in \mathcal{A}^{\mathcal{P}}[S]$ , a contradiction.

Similarly for odd  $b$ , we consider four subsets of size  $\lceil b/2 \rceil$ :  $S_0 = \{p_0, p_2, \dots, p_{b-3}, p_{b-2}\}$ ,  $S'_0 = \{p'_0, p_2, \dots, p_{b-3}, p_{b-2}\}$ ,  $S_1 = \{p_0, p_1, p_3, \dots, p_{b-2}\}$  and  $S'_1 = \{p'_0, p_1, p_3, \dots, p_{b-2}\}$ . Using the same arguments as the *even* case above, we infer that  $S_0, S'_0, S_1, S'_1 \notin \mathcal{A}^{\mathcal{P}}[S]$ . Also it is worth noting that  $S_0 \cup S'_0 \cup S_1 \cup S'_1 = S$ ; in other words, any element of  $S$  will belong to at least one of these four subsets. Again we assume that  $\mathcal{A}^{\mathcal{P}}[S]$  contains a  $b$ -chain. We claim that in such a  $b$ -chain, the only elements that are allowed to be adjacent to  $p_0$  ( $p'_0$  resp.) are  $p'_0$  ( $p_0$  resp.) and  $p_{b-2}$ . Because, for example, if  $p_0$  was adjacent to some  $p_k$  in the  $b$ -chain where  $k \in [1, b-3]$ , then using the *bins* terminology, the bin defined by  $\{p_0, p_k\}$  is not covered by either  $S'_0$  or  $S'_1$  (depending on whether  $k$  is odd or even respectively), thereby arriving at either  $S'_0 \in \mathcal{A}^{\mathcal{P}}[S]$  or  $S'_1 \in \mathcal{A}^{\mathcal{P}}[S]$ , both of which are contradictions. But since  $b > 3$ , such a configuration cannot exist – i.e., no matter how we arrange  $p_0, p'_0$  and  $p_{b-2}$  in the assumed  $b$ -chain, at least one of  $p_0$  or  $p'_0$  has to be adjacent to some  $p_k$  where  $k \in [1, b-3]$ . This concludes the proof.  $\blacktriangleleft$

**Necessary Number of  $b$ -Minicasts.** Another main result of Jaffe et al. [8] is a tight characterization of the quantity  $T_n(t)$  which is defined as the minimum  $m$  such that there exists a general 3-minicast network  $\mathcal{N}$  with  $m$  3-minicast channels that achieves Byzantine agreement while tolerating at most  $t$  corrupted parties – the underlying graph (2-minicast network) of  $\mathcal{N}$  is also assumed to be sufficiently connected, e.g., via a complete set of bilateral channels among the  $n$  parties. To extend their analysis to general  $b$ -minicast networks, for  $b > 3$ , we start with generalizing the definition of  $T_n(\cdot)$  in a higher  $b$ -minicast setting. Similar to [8], in the following we again consider general  $b$ -minicast networks  $\mathcal{N}$  that have a complete set of  $(b-1)$ -minicast channels underneath.

► **Definition 18.** Let  $T_n^b(\mathcal{A})$  denote the minimum number  $m$  such that there exists a general network structure  $\mathcal{N}$  with  $m$   $b$ -minicast channels, satisfying  $\mathcal{N}_{b-1} \subseteq \mathcal{N} \subseteq \mathcal{N}_b$ , that achieves global broadcast for any sender among the  $n$  parties while tolerating the general adversary  $\mathcal{A}$ .

We now provide an upper bound on  $T_n^b(\mathcal{A}^{\mathcal{P}})$ , for  $b > 3$  and any  $b$ -chain adversary  $\mathcal{A}^{\mathcal{P}} \in \mathfrak{A}^{(b)}$ . Again, we note that this is in a similar vein to the analysis in [8] because the threshold adversaries  $n/3 \leq t < n/2$  belong to the class  $\mathfrak{A}^{(3)}$  (see the full version); here we consider the weakest adversaries, a.k.a.  $b$ -chain adversaries, of the class  $\mathfrak{A}^{(b)}$ . The idea behind the upper bound is to explicitly construct a network  $\mathcal{N}$ , of the type described in Definition 18, that satisfies the sufficiency condition of Theorem 13 for broadcast against a  $b$ -chain adversary  $\mathcal{A}^{\mathcal{P}}$ . And for a tighter bound, we would like  $\mathcal{N}$  to have as few  $b$ -minicast channels as possible.

So to construct the network  $\mathcal{N}$ , we start with the complete  $b$ -minicast model and remove  $b$ -minicast channels that are not required to achieve Byzantine broadcast against  $\mathcal{A}^{\mathcal{P}}$ , i.e., minicast channels shared by any subset  $S$  of  $b$  parties where  $\mathcal{A}^{\mathcal{P}}[S]$  is  $b$ -chain free. Specifically in our case, the discarded  $b$ -minicast channels will be of the type described in the proof of Lemma 17: if  $\mathcal{P} = (\mathcal{P}_0, \mathcal{P}_1, \dots, \mathcal{P}_{b-1})$  is the  $b$ -chain corresponding to the adversary  $\mathcal{A}^{\mathcal{P}}$ , then we remove the minicast channel shared by the subset  $S = \{p_0, p'_0, p_1, \dots, p_{b-2}\}$  where  $\{p_0, p'_0\} \subseteq \mathcal{P}_0, p_1 \in \mathcal{P}_1, \dots, p_{b-2} \in \mathcal{P}_{b-2}$ , leaving out  $\mathcal{P}_{b-1}$  (and assuming  $\mathcal{P}_0$  contains more than one party). In fact, we can also consider another type of  $b$ -minicast channels where, for the corresponding subset  $S$ , we pick two parties from  $\mathcal{P}_1$  (if possible) and continue to select

a party from consecutive chain elements, namely  $\mathcal{P}_2, \mathcal{P}_3, \dots, \mathcal{P}_{b-1}$ , leaving out  $\mathcal{P}_0$  this time. And we could also start with two parties from  $\mathcal{P}_0$  and continue picking a party in the reverse direction, i.e., from elements  $\mathcal{P}_{b-1}, \mathcal{P}_{b-2}, \dots, \mathcal{P}_2$  – it is not hard to see that for such a subset  $S$ , the adversary  $\mathcal{A}^{\mathcal{P}}[S]$  is going to be  $b$ -chain free. Considering all these combinations, the resulting number of  $b$ -minicast channels in our network will be an upper bound on  $T_n^b(\mathcal{A}^{\mathcal{P}})$ . For the sake of brevity, let  $\sigma_i = |\mathcal{P}_i|$  with  $\sigma_{i-1} = |\mathcal{P}_{(i-1) \bmod b}|$  and  $\sigma_{i+1} = |\mathcal{P}_{(i+1) \bmod b}|$ . Also let  $\pi_{\mathcal{P}} = \prod_{i=0}^{b-1} \sigma_i$ . Then we have the following:

$$T_n^b(\mathcal{A}^{\mathcal{P}}) \leq \binom{n}{b} - \pi_{\mathcal{P}} \cdot \sum_{i=0}^{b-1} \frac{1}{\sigma_i} \left( \frac{\binom{\sigma_{i-1}}{2}}{\sigma_{i-1}} + \frac{\binom{\sigma_{i+1}}{2}}{\sigma_{i+1}} \right)$$

To compute the above bound, we sum over all unpicked elements  $\mathcal{P}_i$  and choose two parties either from  $\mathcal{P}_{(i-1) \bmod b}$  or  $\mathcal{P}_{(i+1) \bmod b}$ ; this also implies that the set of removed  $b$ -minicast channels w.r.t. each of the above combinations are disjoint, i.e., there is no need to account for any intersections among distinct types of such minicast channels. We believe that the above bound can be made tighter by exhaustively identifying (and removing) additional  $b$ -minicast channels shared by subsets  $S$  of  $b$  parties such that  $\mathcal{A}^{\mathcal{P}}[S]$  is  $b$ -chain free and which do not fall under the type (or its above variants) described in the proof of Lemma 17.

## 6 Conclusions

We identified certain types of partial broadcast channels that are necessary or that suffice for global broadcast to be realized securely against general adversaries on general communication networks. The analysis included proposing a partial ordering over all possible adversary structures with respect to  $n$  parties, which could be of independent interest.

Our results are general enough to characterize such important sets of partial broadcasts of any size  $b$ , against general adversary structures. In particular, this allows us to extend the results of Jaffe et al. [8] related to finding the number of necessary and number of sufficient 3-minicast channels for constructing broadcast to general  $b$ -minicast channels. But at the same time, we note that there is a lot of room for improvement because, in contrast to general 3-minicast networks, we do not (yet) have *tight* necessary and sufficient conditions on general  $b$ -minicast networks – for  $b > 3$  – towards achieving secure broadcast.

So a clear open problem would be to formulate such conditions for higher minicast networks. One approach could be to generalize the tight sufficiency condition of secure broadcast on general 3-minicast networks, as studied in [11, 8], to  $b$ -minicast networks. In the full version, we show that a straightforward extension of the so-called *virtual party emulation* technique – used by [11] for their broadcast protocols in general 3-minicast networks – is not feasible when it comes to  $b$ -minicast channels. Hence it would be interesting to see how one may circumvent this limitation.

On an abstract level, our necessary conditions looked at the  $b$ -chain property of general adversaries on a *global* scale, i.e., with respect to all  $n$  parties. Whereas for our sufficient conditions, we considered the  $b$ -chain condition of adversary structures on a *local* scale, namely w.r.t. subsets of  $b$  parties. Thus, another intriguing open question would be to know whether a middle ground between these two viewpoints could lead to tighter necessary and sufficient conditions on general networks for realizing broadcast. All in all, we hope this paper laid a firm groundwork for more advanced research on broadcast in general communication networks that is secure against general adversaries.

---

**References**

---

- 1 P. Berman, J. A. Garay, and K. J. Perry. Bit optimal distributed consensus. In *Computer Science Research*, pages 313–322, New York, NY, USA, 1992. Plenum Publishing Corporation.
- 2 B. A. Coan and J. L. Welch. Modular construction of a byzantine agreement protocol with optimal message bit complexity. *Information and Computation*, 97:61–85, March 1992.
- 3 J. Considine, M. Fitzi, M. Franklin, L. A. Levin, U. Maurer, and D. Metcalf. Byzantine agreement given partial broadcast. *Journal of Cryptology*, 18(3):191–217, July 2005.
- 4 M. Fitzi and U. Maurer. From partial consistency to global broadcast. In F. Yao, editor, *Proc. 32nd ACM Symposium on Theory of Computing – STOC 2000*, pages 494–503. ACM, May 2000.
- 5 M. Fitzi and U. M. Maurer. Efficient byzantine agreement secure against general adversaries. In S. Kutten, editor, *DISC*, volume 1499 of *Lecture Notes in Computer Science*, pages 134–148. Springer, 1998.
- 6 O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *Proceedings of the 19th annual ACM symposium on theory of computing, STOC '87*, pages 218–229, New York, NY, USA, 1987. ACM.
- 7 Martin Hirt and Ueli Maurer. Complete characterization of adversaries tolerable in secure multi-party computation. In *PODC*, volume 97, pages 25–34, 1997.
- 8 A. Jaffe, T. Moscibroda, and S. Sen. On the price of equivocation in byzantine agreement. In D. Kowalski and A. Panconesi, editors, *ACM Symposium on Principles of Distributed Computing, PODC '12, Funchal, Madeira, Portugal, July 16-18, 2012*, pages 309–318. ACM, 2012.
- 9 Raykov P. Broadcast from minicast secure against general adversaries. In M. M. Halldórsson, K. Iwama, N. Kobayashi, and B. Speckmann, editors, *ICALP 2015: 42nd International Colloquium on Automata, Languages and Programming, Part II, Kyoto, Japan, July 6-10, 2015*, volume 9135 of *Lecture Notes in Computer Science*, pages 701–712. Springer, Berlin, Germany, 2015.
- 10 M. C. Pease, R. E. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.
- 11 D. V. S. Ravikant, M. Venkitasubramaniam, V. Srikanth, K. Srinathan, and C. P. Rangan. On byzantine agreement over (2,3)-uniform hypergraphs. In R. Guerraoui, editor, *Distributed Computing, 18th International Conference, DISC 2004, Amsterdam, The Netherlands, October 4-7, 2004, Proceedings*, volume 3274 of *Lecture Notes in Computer Science*, pages 450–464. Springer, 2004.
- 12 A. C. Yao. Protocols for secure computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science, SFCS '82*, pages 160–164, Washington, DC, USA, 1982. IEEE Computer Society.

# Maximally Resilient Replacement Paths for a Family of Product Graphs

Mahmoud Parham 

University of Vienna, Faculty of Computer Science, Vienna, Austria  
mahmoud.parham@univie.ac.at

Klaus-Tycho Foerster 

University of Vienna, Faculty of Computer Science, Vienna, Austria  
klaus-tycho.foerster@univie.ac.at

Petar Kusic 

University of Vienna, Faculty of Computer Science, Vienna, Austria  
petar.kusic@univie.ac.at

Stefan Schmid 

University of Vienna, Faculty of Computer Science, Vienna, Austria  
stefan\_schmid@univie.ac.at

---

## Abstract

---

Modern communication networks support fast path restoration mechanisms which allow to reroute traffic in case of (possibly multiple) link failures, in a completely *decentralized* manner and without requiring global route reconvergence. However, devising resilient path restoration algorithms is challenging as these algorithms need to be inherently *local*. Furthermore, the resulting failover paths often have to fulfill additional requirements related to the policy and function implemented by the network, such as the traversal of certain waypoints (e.g., a firewall).

This paper presents local algorithms which ensure a maximally resilient path restoration for a large family of product graphs, including the widely used tori and generalized hypercube topologies. Our algorithms provably ensure that even under multiple link failures, traffic is rerouted to the other endpoint of every failed link whenever possible (i.e. *detouring* failed links), enforcing waypoints and hence accounting for the network policy. The algorithms are particularly well-suited for emerging segment routing networks based on label stacks.

**2012 ACM Subject Classification** Networks → Routing protocols; Computer systems organization → Dependable and fault-tolerant systems and networks; Mathematics of computing → Graph algorithms

**Keywords and phrases** Product Graphs, Resilience, Failures, Routing

**Digital Object Identifier** 10.4230/LIPICs.OPODIS.2020.26

**Funding** Research (in part) supported by the Vienna Science and Technology Fund (WWTF) project WHATIF, ICT19-045, 2020-2024.

## 1 Introduction

Communication networks have become a critical infrastructure of our society. With the increasing size of these networks, however, link failures are more common [2, 10], which emphasizes the need for networks that provide a reliable connectivity even in failure scenarios, by quickly rerouting traffic. As a global re-computation (and distribution) of routes after failures is slow [21], most modern communication networks come with fast *local* path restoration mechanisms: conditional failover rules are *pre-computed*, and take effect in case of link failures *incident* to a given router.



© Mahmoud Parham, Klaus-Tycho Foerster, Petar Kusic, and Stefan Schmid;  
licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 26; pp. 26:1–26:16

Leibniz International Proceedings in Informatics



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

Devising algorithms for such path restoration mechanisms is challenging, as the failover rules need to be *(statically) pre-defined* and can only depend on the *local* failures; at the same time, the mechanism should tolerate multiple or ideally, a *maximal* number of failures (as long as the underlying network is still connected), no matter where these failures may occur. Furthermore, besides merely re-establishing connectivity, reliable networks often must also account for additional network properties when rerouting traffic: unintended failover routes may disrupt network services or even violate network policies. In particular, it is often important that a flow, along its route from  $s$  to  $t$ , visits certain policy and network function critical “waypoints”, e.g., a firewall or an intrusion detection system, even if failures occur.

Today, little is known about how to provably ensure a high resiliency under multiple failures while preserving visits to waypoints. This paper is motivated by this gap. In particular, we investigate local path restoration algorithms which do not only provide a maximal resilience to link failures, but also never “skip” nodes: rather, traffic is rerouted around failed links individually, hence *enforcing waypoints* [1].

## 1.1 Related Work

**Motivation.** Resilient routing is a common feature of most modern communications networks [6], and the topic has already received much interest in the literature. However, most prior research on static fast rerouting aims at restoring connectivity to the final destination, without considering waypoint properties as in our work. Such waypoint preservation is motivated by the advent of (virtualized [11]) middleboxes [4], respectively *local protection schemes* in Multiprotocol Label Switching (MPLS) terminology [26], and by the recent emergence of Segment Routing (SR), where routing is based off label stacks – more precisely by the label on top of the stack [24], which is treated as the next routing destination.

**Path restoration.** Only little is known today about static fast rerouting under multiple failures, while preserving waypoints. In TI-MFA [16], it has been shown that existing solutions for SR fast failover, based on TI-LFA [20], do not work in the presence of two or more failures. However, TI-MFA [16] and non-SR predecessors [22] rely on failure-carrying packets, which is undesirable as discussed before and we overcome in the current paper.

For the case of two failures, heuristics [9] exist, but they do not provide any formal protection guarantees, except for torus graphs [23]. Beyond a single failure [20] in general and two failures on the torus [23], we are not aware of any approaches that work in our model, except for a recent work on binary hypercubes [17]. However, it is not clear how to extend [17] to e.g. generalized hypercubes, and the approach followed in this paper presents a more generic scheme for the Cartesian product of *any* set of base graphs, as long as “well-structured” base graph schemes are provided.

**Connectivity restoration without waypoints.** Static fast failover mechanisms without waypoints are investigated by Chiesa et al. [5, 7, 8] leveraging arc-disjoint network decompositions, also by Elhourani et al. [10], Stephens et al. [27, 28], and Schmid et al. [3, 14, 15, 18, 19, 25]. Beyond that, the concept of perfect resilience (any number of failures) is investigated in [12, 13, 29]. Even though it is possible to provide  $\Omega(k)$ -resilience in  $k$ -connected graphs, this guarantee pertains only to reaching the destination, and does not transfer to link protection.



## 1.2 Contributions

We initiate the study of local (i.e., *immediate*) path restoration algorithms on product graphs, an important class of network topologies. More specifically, our algorithms are 1) resilient to a maximum number of failures (i.e., are *maximally robust*), 2) respect the (waypoint) path traversal of the original route (by detouring failed links), and 3) are compatible with current technologies, and in particular with emerging segment routing networks [24]: our algorithms do not require packets to carry failure information, routing tables are static, and forwarding just depends on the packet's top-of-the-stack destination label and the incident link failures.

Our main result is an efficient scheme that can provide maximally resilient backup paths for arbitrary Cartesian product of given base graphs, as long as “well-structured” schemes are provided for the base graphs. Using complete graphs, paths, and cycles as base graphs, we can generate maximally resilient schemes for additional important network topologies such as grids, tori, and generalized hypercubes.

## 1.3 Organization

The remainder of this paper is organized as follows. We first introduce necessary model preliminaries in Section 2, followed by our main result in Section 3, where we provide a general scheme to compute maximally resilient path restoration schemes for product graphs. We then show how our scheme can be leveraged for specific graph classes in Section 4, for the selected examples of complete graphs, generalized hypercubes, grids, and torus graphs. We conclude our study in Section 5 with a few open questions.

## 2 Preliminaries

We consider undirected graphs  $G = (V, E)$  where  $V$  is the set of *nodes* and  $E$  is the set of *links* connecting nodes.

► **Definition 1.** A backup path (a.k.a. replacement path) for a link  $\ell \in E$  is a simple path that connects the endpoint of the link  $\ell$ . Let  $\mathcal{P}$  be the set of all backup paths in a graph. An injective function  $BP_G : E \rightarrow \mathcal{P}$  that maps each link to one of its backup paths is a backup path scheme.

We may drop the subscript when the graph  $G$  is clear from the context. When a packet arrives at a node and the next link on its path is some failed link  $\ell_1$ , the node (i.e., router) immediately reroutes the packet along the backup path of  $\ell_1$ , given by  $BP(\ell_1)$ . The packet may encounter a second failed link  $\ell_2 \in BP(\ell_1)$ . Now assume  $\ell_1 \in BP(\ell_2)$ . The packet loops between the two links indefinitely as one link lies on the BP of the other. To this end, we need to characterize backup paths that do not induce such infinite forwarding loops under any subset of simultaneous link failures restricted only in cardinality. Before that, we formalize the actual route that a packet takes under the “failure scenario”  $L$ .

► **Definition 2.** Given any subset of links  $L \subset E$ , a detour route around a link  $\ell \in L$ , denoted by  $R_G(\ell, L)$ , is obtained by recursively replacing each link in  $BP_G(\ell) \cap L$  with its respective detour route. Precisely,

$$R_G(\ell, L) = (BP_G(\ell) \setminus L) \cup \bigcup_{\ell' \in BP_G(\ell) \cap L} R_G(\ell', L). \quad (1)$$

Moreover, 1)  $BP_G$  is resilient under the failure scenario  $L$  if and only if  $\forall \ell \in L$ , the detour  $R_G(\ell, L)$  exists, i.e., the recursion terminates, and

2)  $BP_G$  is  $f$ -resilient if and only if it is resilient under every  $L \subset E$  s.t.  $|L| \leq f$ .

In words, when a packet's next hop is across the failed link  $\ell \in L$ , it gets rerouted along the route  $R_G(\ell, L)$  which ends at the other endpoint of  $\ell$  hence evading all failed links. A BP scheme is  $f$ -resilient if for every subset of up to  $f$  failed links, replacing each failed link with its backup path produces a route that excludes failed links. The replacement process from a packet's perspective occurs recursively as in (1). A packet ends up in a loop permanently when it encounters a failed link for which the detour (1) does not exist. Then, the scheme is  $f$ -resilient if a packet that encounters a failed link reaches the other endpoint of the link by traversing the BP of that link and the BP of any consequent failed link that it encounters along the way.

Definition 2 implies that we cannot have a resiliency higher than graph connectivity, since  $L$  may simply consist of all links incident to one node which makes a detour impossible.

► **Definition 3.** An  $f$ -resilient backup path scheme  $BP_G$  is maximally resilient if and only if there is no  $(f + 1)$ -resilient scheme.

Next, we introduce the notion of “dependency” on which we establish some key definitions used widely in the analysis of resiliency in our proofs.

► **Definition 4.** We say there is a dependency relation  $\ell \rightarrow \ell'$  if and only if the link  $\ell$  includes the link  $\ell'$  on its backup path, i.e.,  $\ell' \in BP_G(\ell)$ . We represent all dependency relations as a directed dependency graph  $\mathcal{D}(BP_G)$  with vertices  $\{v_\ell \mid \ell \in G\}$  and arcs  $\{(v_{\ell_1}, v_{\ell_2}) \mid \ell_1 \rightarrow \ell_2\}$ .  $BP_G$  induces the dependency graph  $\mathcal{D}(BP_G)$ .

We denote a dependency arc  $(v_{\ell_1}, v_{\ell_2})$  by  $(\ell_1, \ell_2)$  for simplicity. Any backup path scheme  $BP_G$  induces cycles in  $\mathcal{D}(BP_G)$ , as otherwise there is a link without any BP assigned to it. We refer to one such cycle as *cycle of dependencies* or CoD for short. Similarly, we define *path of dependency* or PoD for short.

Observe that a CoD captures a failure scenario that leads to a permanent loop. Rewording Definition 2,  $BP_G$  is  $f$ -resilient if and only if every CoD is longer than  $f$ , i.e., it consists of at least  $f + 1$  dependency arcs. Hence, CoDs with the shortest length determine the resiliency and we refer to them as *min-CoDs*.

Next, we introduce some additional notations and definitions based on Definition 4. Let  $CoD(v)$  denote the CoD over links incident to  $v \in V$ . We consider maximally resilient schemes for special regular graphs which implies  $CoD(v)$  is unique. Note that non-incident links may induce (min-)CoDs as well. We focus on special regular graphs and resiliency thresholds that are maximal for the connectivity (or the degree) of the those graphs. Then, a min-CoD cannot be shorter than the degree of the respective regular graph, which implies  $CoD(v)$  is unique for every node  $v$ .

In Section 3, we present a backup path scheme for certain  $k$ -dimensional *product graphs*, by generalizing the solution presented in [17] on binary hypercubes (*BHC*). A  $k$ -dimensional BHC is the Cartesian product of any set of BHCs where dimensions add up to  $k$ . A product graph  $\mathcal{G}$  is the Cartesian product of *base graphs* in  $\{g^1, \dots, g^k\}$ . That is,  $\mathcal{G} = \prod_{d \in [k]} g^d$  where  $\prod$  denotes the Cartesian product and each  $g^d$  is the base graph *in dimension*  $d$ . Let  $n_d := |V[g^d]|$ ,  $d \in [k]$  denote the order of  $g^d$ . Nodes in a product graph are represented as  $k$ -tuples  $(a_k, \dots, a_1)$  where  $\forall d \in [k] : 0 \leq a_d < n_d$ . Likewise, we assume labels  $(a_k, \dots, a_{d-1}, *, a_{d+1}, \dots, a_1)$  for links where their endpoint nodes differ in their  $d$ th digit (i.e.,  $d$ th component) which is represented by the “\*”.

### 3 Resiliency for Cartesian Product

We now introduce an algorithm to compute a maximally resilient scheme for special product graphs. More specifically, the algorithm takes the scheme of each base graph and combines them in a way that yields a scheme for the Cartesian product of those base graphs. However, it requires each individual scheme to possess some structural properties. We begin with the characterization of these properties.

We can *break* a CoD open into a PoD by removing one of its arcs, which is achieved by removing the head link of an arc from the BP of its tail link.

► **Definition 5.** *An  $r$ -resilient backup path scheme  $BP_G$  is well-structured if and only if there is a set of boundary links  $L^*$  that for every node  $v$  contains a unique link incident to  $v$ , satisfying the following conditions.*

1. *There is a unique CoD  $C^*$  that consists only of links in  $L^*$ .*
2. *The following procedure breaks all CoDs.*
  - a. *For every link  $\ell \notin L^*$  s.t.  $BP_G(\ell) \cap L^* \neq \emptyset$ ;*
    - i. *There are exactly two nodes  $x_1$  and  $x_2$  on  $BP(\ell)$ , s.t.  $L^*(x_1), L^*(x_2) \in BP(\ell)$ .*
    - ii. *Remove every link of  $BP(\ell)$  between  $x_1$  and  $x_2$ , i.e. the subpath  $BP(\ell)[x_1, x_2]$ .*
  - b. *To break  $C^*$ , pick one arc  $(\ell', \ell^*) \in C^*$  arbitrarily and remove  $\ell^*$  from  $BP_G(\ell')$ .*
3. *At least  $r$  arcs are left in every CoD (not removed at 2(a)ii).*

Intuitively, these conditions mandate a choice of  $L^*$  that for every CoD, the packet that realizes the CoD traverses a boundary link. Removing the arc headed at such link breaks the CoD open into a PoD. We refer to such arc as a *feedback arc*. Later, we close the PoD into a new CoD that is induced by the scheme of a product graph for which  $G$  is a “base graph”.

Concretely, Definition 5 constrains the set  $L^*$  in a way that for every CoD one of the following two cases must apply. Case 1. The CoD may contain an arc headed to a link in  $L^*$ . Then removing the head link from the BP of the tail link is sufficient to break the CoD. Case 2. The CoD may not contain any link in  $L^*$  as the tail or head of some arc, but it contains an arc  $(\ell_1, \ell_2), \ell_2 \notin L^*$  that the packet departing from either endpoints of  $\ell_1$ , traversing  $BP_G(\ell_1)$ , has to traverse some link in  $L^*$  before reaching  $\ell_2$ . The procedure (at line 5.2(a)ii), removes not only links of  $L^*$  from the BP but also the link  $\ell_2$ , since it lies between  $x_1$  and  $x_2$ . Note that Case 1 applies also to the unique CoD  $C^*$  which is handled separately at 5.2b.

Next, we establish a lemma that constructs a walk on all nodes of  $G$ , using a given BP scheme and the corresponding set of boundary links.

► **Lemma 6.** *Assume a well-structured scheme  $BP_G$  and a set of links  $L^*$  satisfying Definition 5 are given. There exists a closed walk  $W$  on all nodes of  $G$  that 1) visits each node  $v \in G$  immediately before traversing the link  $L^*(v)$ , and 2) links in  $L^*$  are traversed in the same order they are traversed by  $C^*$ .*

**Proof.** The following procedure marks every node in  $G$  with FINISHED as soon as a visit to  $v$  is followed by walking the link  $L^*(v)$ .

1.  $W = \emptyset$ .
2. Let  $w_0 := v$ . Initialize with the last traversed boundary link  $\ell^* = L^*(w_0)$ . Let  $\{w_0, w_1\} := \ell^*$ , then initialize the walk  $W = [w_0, w_1]$ .
3. Repeat:
  - a. Assume  $W = [w_0, w_1, \dots, w_t]$  is the current walk,  $L^*(w_t) = \{w_t, u\}$  and let  $\ell'_{w_t} := \{w_t, u'\} \in BP_G(\ell^*), u' \neq w_{t-1}$ .

- b. If  $w_{t-1} = u \wedge w_t \neq w_{t-2}$  then  $w_{t+1} = u$ .
- c. Else,  $w_{t+1} = u'$ .
- d. If  $w_{t+1} = u$  then  $\ell^* = \ell_{w_t}$  and mark  $w_t$  with FINISHED.
- e. If  $w_t = w_0 \wedge \{w_0, w_1\} \in BP_G(\ell^*)$  then Break.

The walk  $W$  begins with the link  $L^*(w_0)$ . Then it proceeds to the next link on the backup path of the last traversed link  $\ell^* \in L^*$  at Line 3c (initially  $\ell^* = \ell_{w_0}$ ), or it traverses the recently walked link  $\{w_{t-1}, w_t\}$  in the opposite direction at Line 3b (i.e., from  $w_t$  to  $w_{t-1}$ ). By assumption, any  $\ell \in L^*$  is on the backup path of some  $\ell' \in L^*$  and  $(\ell', \ell) \in \mathcal{C}_{BP_G}^*$ . Therefore, the loop at Line 3 reaches an iteration where the last traversed  $\ell^* \in L^*$  includes  $L^*(w_0)$  on its backup path, which breaks the loop at Line 3e. The last visited node must be  $w_0$  implying  $W$  is a closed walk. Whenever  $W$  reaches a node  $w_t$  and  $L^*(w_t)$  is on the backup path of the last traversed  $\ell^* \in L^*$ , then it next traverses  $L^*(w_t)$  for the first time at Line 3c in one direction, or for the second time at Line 3b in the reverse direction. In either case,  $L^*(w_t)$  is walked immediately after a (FINISHED) visit to  $w_t$ . At the end, both endpoints of every link in  $L^*$  are marked FINISHED and since  $\bigcup_{\ell \in L^*} \ell = V[G]$ , all nodes are marked FINISHED.  $\blacktriangleleft$

We will use the walk in the construction of the scheme for a multi-dimensional graph where  $G$  is the base graph in some dimension. The walk is used to guide backup paths of links in other dimensions when they need to traverse the dimension of  $G$ .

### 3.1 The Construction

For every base graph  $g^d$ , we assign node labels  $0, \dots, n_d - 1$  such that nodes are ordered as they are FINISHED in Lemma 6. I.e., the first node FINISHED gets 0, the second one gets 1 and so on. Assume, for each  $g^d \in \mathcal{G}$ , a well-structured,  $r_d$ -resilient backup path scheme  $BP_{g^d}$  together with a boundary set  $L_{BP_{g^d}}^* \subseteq E[g^d]$  is given. Let us fix a circular order over base graphs, e.g.,  $g^1, \dots, g^d$ . A node  $v := (a_1, \dots, a_k) \in \mathcal{G}$  corresponds to the  $a_d$ th node in the  $d$ th base graph  $g^d$ ,  $d \in [k]$ .

Let  $inc_d(1, \dots, a_k)$  denote the (successor) function that takes a node in  $\mathcal{G}$ , increments the  $d$ th digit, applies any carry flag rightward rotating left, and discards any carry back to the  $d$ th digit. Observe that for a fixed  $d \in [k]$ , the function  $inc_{d+1}$  defines a total order over all instances of  $g^d$ . We denote the  $i$ th instance by  $g_i^d$ . We write  $g_i^d$  (instead of  $g^d$ ) only when we refer to a specific  $g^d$ -instance. Similarly,  $\ell \in \mathcal{G}$  is a  $g^d$ -link if it is an instance of a link in  $g^d$ .

Let  $v_i^d(x)$  denote the mapping  $V[g^d] \mapsto V[g_i^d] \subseteq V[\mathcal{G}]$ , where  $v_i^d(x)$  is the  $i$ th instance of the node  $x \in g^d$ . Then,  $v_{i+1}^d(x) = inc_{d+1}(v_i^d(x))$ . Similarly, for a path (i.e., subset) of nodes  $P$ , we have  $v_i^d(P) = \cup_{v \in P} v_i^d(v)$ . We use  $v_i^d$  whenever the node  $x$  is not relevant to the context. Next, we compute a path  $P^*(v_i^d) = \{v_i^d, \dots, v_{i+1}^d\}$ , that connects  $v_i^d$  and  $v_{i+1}^d$  in  $\mathcal{G}$  through the sequence of base graphs  $g^{d+1}, g^{d+2}, \dots$ . The intermediate nodes are determined by digits incremented during the operation  $inc_{d+1}(v_i^d)$ . Algorithm 1 depicts this procedure.

We initialize the scheme for every  $g^d$ -instance with a copy of  $BP_{g^d}$ , i.e.,  $\forall i : BP_{g_i^d} = BP_{g^d}$ . Then, we integrate  $BP_{g_i^d}$  into  $BP_{\mathcal{G}}$  by extending backup paths of links that contain or traverse a boundary link, i.e., links that are tail of some feedback arc. Consider any feedback arc  $(\ell, \ell') \in \mathcal{A}_{BP_{g_i^d}}(\mathcal{C})$ . Since  $\ell' \in BP_{g_i^d}(\ell)$ , we can break  $\mathcal{C}$  by extending  $BP_{g_i^d}(\ell)$  into a backup path that does not traverse  $\ell'$  (i.e., detours  $\ell'$ ). We detour  $\ell' = \{x_1, x_2\}$  via a pair of walks through  $g_i^{d+1}, g_i^{d+1}, \dots$  that reaches the next instance of  $g_i^d$ , i.e., the instance given by  $inc_{d+1}$ . That is, the paths  $P^*(v_i^d(x_1))$  and  $P^*(v_i^d(x_2))$ . By reconnecting  $v_{i+1}^d(x_1)$  and  $v_{i+1}^d(x_2)$  through  $g_{i+1}^d$ , we finish the construction of the extended backup path. In Algorithm 2, we use notations and constructions defined so far to describe the integration of all  $BP_{g_i^d}$ 's into one scheme  $BP_{\mathcal{G}}$ .

■ **Algorithm 1** Construction of  $P^*(v_i^d), v_i^d = (a_0, \dots, a_{k-1})$ .

---

```

1: function  $P^*(v_i^d)$ 
2:    $P = \{v_i^d\}, v = v_i^d, d' = d + 1, carry = 1$  ▷ initialize
3:   while  $carry > 0 \wedge d' \neq d$  do ▷ emulating  $inc_{d+1}(v)$ 
4:     if  $a_{d'} < n_{d'} - 1$  then
5:        $v[d'] = v[d'] + 1, carry = 0$  ▷ increment the  $d'$ th digit
6:     else
7:        $v[d'] = 0, carry = 1$ 
8:        $d' = (d' + 1) \pmod{k}$  ▷ move to the next digit, rotating left
9:      $P = P \cup \{v\}$  ▷ append  $v$  to  $P$ 
10:  return  $P$ 

```

---

■ **Algorithm 2** Construction of  $BP_G$ .

---

```

1: Initialize  $BP_G = \emptyset$ 
2: for every  $d \in [k]$  and all instances  $g_i^d$  do
3:    $BP_{g_i^d} = \text{FORBASEGRAPH}(d, i)$ 
4:  $BP_G = \bigcup_{d \in [k], i} BP_{g_i^d}$ 
5: function  $\text{FORBASEGRAPH}(d, i)$ 
6:   Initialize  $BP_{g_i^d} = BP_{g^d}$ , relabel all nodes from  $x \in g^d$  to  $v_i^d[x] \in g_i^d$ .
7:   Let  $L_i^d := L^*$  of  $BP_{g_i^d}$  (Definition 5)
8:   for every  $\ell \in g_i^d, \ell \notin L_i^d$  s.t.  $BP_{g_i^d}(\ell) \cap L_i^d \neq \emptyset$  do ▷ Definition 5.2a
9:     Let  $x_1$  and  $x_2$  be nodes as specified in Definition 5.2(a)i. ▷ detour points
10:     $S := BP_{g_i^d}(\ell)[x_1, x_2]$  ▷ the part of BP to be removed
11:     $S^* := inc_{d+1}(S)$  ▷ the copy of  $S$  in the next  $g^d$ -instance  $g_{i+1}^d$ 
12:    Compute  $P^*(x_1)$  and  $P^*(x_2)$  ▷ Algorithm 1
13:     $P'_\ell := (P_\ell \setminus \{S\}) \cup \{S^*\} \cup P^*(x_1) \cup P^*(x_2)$ 
14:     $BP_{g_i^d}(\ell) = P'_\ell$ 
15:  return  $BP_{g_i^d}(\ell)$ 

```

---

► **Definition 7.** Let  $\ell_1 := \{u, v\} \in g_i^d, \ell_2 := \{u', v'\} \in g_j^d, j \neq i$ . We say that the dependency arc  $(\ell_1, \ell_2)$  traverses the base graph  $g^d, d \neq d'$  if and only if  $\ell_1$  and  $\ell_2$  differ in their  $d$ th digits. Moreover, if the  $d$ th digit from  $\ell_1$  to  $\ell_2$  increases by 1 then we say the arc traverses  $g^d$  in uphill direction. Otherwise the  $d$ th digits resets to zero and the arc traverses  $g^d$  in downhill direction.

Restating Definition 7, two packets departing from the two endpoints of  $\ell_1$  traveling on the backup path of  $\ell_1$  together traverse a pair of links in two  $g^d$ -instances (symmetrically), before reaching  $\ell_2 \in BP_{g_i^d}(\ell_1)$ . The pair of  $g^d$ -links are distinct instances of the same link in  $g^d$  and they are traversed in the same direction due to the symmetric construction of the pair of paths at Line 2.12. That is, either towards their higher endpoint (i.e. larger  $d$ th digit), which we refer to as the uphill direction, or the opposite (downhill) direction.

► **Definition 8.** We say an arc  $(\ell_1, \ell_2), \ell_1 \in g_i^d, \ell_2 \in g_j^d$  crosses  $g^d$  if the two links belong to different base graphs, i.e.  $d' \neq d$ , or both are in the same  $g^d$ -instance, i.e.  $d = d'$  and  $i = j$ .

Similarly, we say a PoD (CoD) traverses or crosses  $g^d$  if it includes an arc that, respectively, traverses or crosses  $g^d$ . Therefore, if a PoD does not cross  $g^d$ -link then it means it does not contain any  $g^d$ -link as the head of an arc. We emphasize that by construction, an arc either crosses or traverses a base graph  $g^d$ .

► **Definition 9.** *An arc  $(\ell_1, \ell_2) \in \mathcal{C}$  is the contribution of  $g^d$  in one these cases: it crosses  $g^d$ , it traverses  $g^d$  in the uphill direction, or  $\ell_2$  is a  $g^d$ -link and the arc traverses all other dimensions in the downhill direction.*

By Definition 9 every arc is the contribution of a unique base graph.

### 3.2 Analysis of Resiliency

We begin with a series of lemmas that show each base graph contributes its resiliency to the resiliency of  $BP_{\mathcal{G}}$ .

► **Lemma 10.** *Let  $P$  be a PoD induced by  $BP_{\mathcal{G}}$  that traverses  $g^d$  in the uphill direction at least once and it does not cross  $g^d$ . Then, there exists a PoD  $\tilde{P}$  induced by  $BP_{g^d}$  that consists of the links in  $L_{BP_{g^d}}^*$  that are traversed by  $P$  s.t.  $|P| \geq |\tilde{P}|$ .*

We defer the proof to the appendix due to space constraint.

**Proof.** We have  $|\mathcal{C}| \geq |\tilde{\mathcal{C}}|$  by applying Lemma 10. Then the claim follows because of the assumption that  $BP_{g^d}$  is  $r_d$ -resilient, which directly implies  $|\tilde{\mathcal{C}}| \geq r_d + 1$ . ◀

► **Lemma 11.** *Let  $P := \{(\ell_{first}, \ell_1), \dots, (\ell_s, \ell_{last})\}$  be a PoD induced by  $BP_{\mathcal{G}}$ . Assume  $\ell_{first} \in g_i^d$  and  $\ell_{last} \in g_j^d$  are the only  $g^d$ -links on  $P$  for some  $i$  and  $j$ . Let  $\ell'_{first}, \ell'_{last} \in g^d$  be the corresponding links in  $g^d$ . Then there exists a PoD  $\tilde{P}$  induced by  $BP_{g^d}$  that begins with  $\ell'_{first}$  and ends at  $\ell'_{last}$  s.t.  $|P| \geq |\tilde{P}|$ .*

**Proof.** By assumption,  $P$  begins with an arc tailed at  $\ell_{first} \in g_i^d$ . Let  $(\ell_{first}, \ell')$  be the feedback arc induced by  $BP_{g_i^d}$  that is picked at Line 2.9 and then is handled by detouring a boundary link  $\ell' \in L_{g_i^d}^*$  via  $g_{i+1}^d$  at Lines 2.9 to 2.14. Let  $A \subseteq P$  be the set of arcs in  $P$  that traverse  $g^d$  in the uphill direction. Note the  $d$ th digit changes only along arcs in  $A$  and remains unchanged along arcs  $P \setminus A$ . We construct a PoD  $\tilde{P}$  over a subset of boundary links in  $L_{g^d}^*$ , as follows. The first arc in  $\tilde{P}$  is  $(\ell_{first}, \ell')$ . With each arc in  $A$ , the  $d$ th digit increases by 1 from its tail to its head. Recall that the value of this digit is a node label in  $g^d$ , and an increment by 1 corresponds to traversing a boundary link of  $g^d$ . Consider arcs in  $A$  sorted in the order they appear in  $P$ . Let  $\ell^* \in L_{BP_{g^d}}^*$  be the boundary link traversed by the first arc in  $A$  (possibly,  $\ell^* = \ell'$ ). Let  $P' := P \setminus \{(\ell_{first}, \ell_1), (\ell_s, \ell_{last})\}$ . By assumption,  $P'$  does not cross  $g^d$  and therefore it begins at  $\ell^*$  and ends at  $\ell^{**}$ , the boundary link traversed by the last arc in  $A$ . we consider two cases.

Case i)  $\ell_{last}$  is a boundary link, i.e.,  $\ell_{last} \in L_{g^d}^*$ , then we apply Lemma 10 to  $P'$  and we obtain a PoD  $P''$ ,  $|P''| \leq |P'|$ , over the boundary links traversed by  $A$ . (1) Due to Line 2.12 and Lemma 6.2, arcs in  $A$  traverse boundary links of  $BP_{g^d}$  in the same order they appear in  $\mathcal{C}_{BP_{g^d}}^*$ . (2) The  $d$ th digit does not change, from the head of the last arc in  $A$  until the arc headed at  $\ell_s$ . Combining (1) and (2) implies that  $\ell_{last}$  succeeds  $\ell^{**}$  in this ordering and therefore  $(\ell^{**}, \ell_{last}) \in \mathcal{C}_{BP_{g^d}}^*$  is an arc induced by  $BP_{g^d}$ . Thus,  $\tilde{P} := \{(\ell_{first}, \ell^*)\} \cup P'' \cup \{(\ell^{**}, \ell_{last})\}$  is a PoD (induced by  $BP_{g^d}$ ) and  $|P| = |P'| + 2 \geq |P''| + 2 = |\tilde{P}|$ , which satisfies the lemma.



Case ii)  $\ell_{last}$  is not a boundary link, i.e.,  $\ell_{last} \notin L_{g^d}^*$ . Let  $w_t$  the value of the  $d$ th digit at  $\ell_s$ . The walk  $W_{BP_{g^d}}$  from Lemma 6 visits the node  $w_t \in g^d$  immediately before traversing the incident boundary link  $\ell^{**} := L_{g^d}^*(w_t)$  (Line 6.3d). The pair of paths computed at Line 2.12 traverse nodes of  $g^d$  (i.e., values of the  $d$ th digits along the paths) in the same order as they are walked on by  $W_{BP_{g^d}}$ . This means that  $BP_{\mathcal{G}}(\ell_s)$  traverses (some two instances of)  $\ell^{**}$  before any other link in  $g^d$ , in particular, before  $\ell_{last}$ . Therefore  $\ell^{**} \in BP_{\mathcal{G}}(\ell_s)$  and  $(\ell_s, \ell^{**})$  is an arc induced by  $BP_{\mathcal{G}}$ . Then,  $P' := P \setminus \{(\ell_s, \ell_{last})\} \cup \{(\ell_s, \ell^{**})\}$  is a PoD as well. By Lemma 6.3, the walk  $W_{BP_{g^d}}$ , after traversing  $\ell^{**}$ , walks on  $BP_{g^d}(\ell^{**})$  until the next boundary link is reached. Hence,  $\ell_{last}$  is on this backup path and  $(\ell^{**}, \ell_{last})$  is an arc induced by  $BP_{g^d}$ . is a PoD induced by  $g^d$ . Now, similarly to the case (i), we remove the first and the last arcs in  $P'$  and obtain a PoD  $P''$  that does not cross  $g^d$ . By applying Lemma 10 to  $P''$ , we obtain a PoD  $P^*$  induced by  $g^d$  s.t.  $|P^*| \leq |P''|$ . Thus,  $\tilde{P} := \{(\ell_{first}, \ell^*)\} \cup P^* \cup \{(\ell^{**}, \ell_{last})\}$  is a PoD induced by  $g^d$  and  $|P| = |P'| = |P''| + 2 \geq |P^*| + 2 = |\tilde{P}|$ , which concludes the lemma.  $\blacktriangleleft$

► **Theorem 12.** *The backup path scheme  $BP_{\mathcal{G}}$  is  $(\Delta - 1)$ -resilient where  $\Delta = \sum_{d \in [k]} (r_d + 1)$ .*

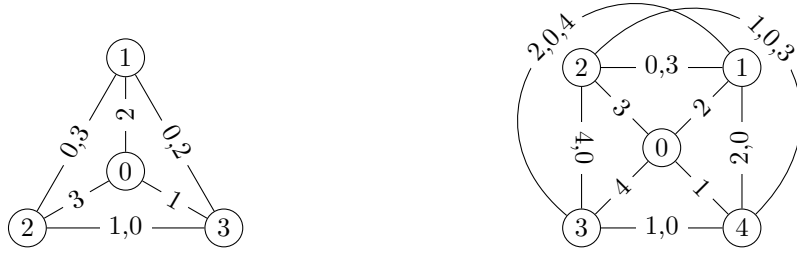
**Proof of Theorem 12.** Consider any CoD  $\mathcal{C}$  induced by  $BP_{\mathcal{G}}$ . We shrink  $\mathcal{G}$  down to a single instance of  $g^d$  denoted by  $\tilde{g}^d$ . To this end, we map all nodes in  $\mathcal{G}$  with equal  $d$ th digit, to one node  $s \in \tilde{g}^d$ . As a result, links between nodes with equal  $d$ th digits merge into a single node, which transforms them into loop links. We remove all arcs having a loop link as an endpoint and denote the remaining arcs by  $\mathcal{C}'$ . Since  $g^d$  is  $r_d$ -resilient,  $g^d$  contributes up to  $r_d + 1$  arcs to  $\mathcal{C}$ ; we argue that the contribution is exactly  $r_d + 1$  arcs.

If  $\mathcal{C}$  consists of  $g^d$ -links only (i.e., endpoints of every arc in  $\mathcal{C}$  have different  $d$ th digits), then all arcs in  $\mathcal{C}$  are preserved (i.e. not removed) after the transformation, which implies  $\mathcal{C}'$  is a CoD in  $\tilde{g}^d$  and  $|\mathcal{C}| \geq |\mathcal{C}'| \geq r_d + 1$ . However, some arcs in  $\mathcal{C}'$  are projection of arcs in  $\mathcal{C}$  that are not the contribution of  $g^d$  (Definition 9). They traverse some  $g^{d'}$ ,  $d' \neq d$  in the uphill direction and hence are exclusively the contribution of  $g^{d'}$ . These are the same arcs eliminated at Line 5.2(a)ii. Definition 5.3 guarantees at least  $r_d$  non-eliminated arcs left which implies at least  $r_d + 1$  arcs in  $\mathcal{C}'$  cross  $g^d$  and are its contribution. There must be one arc that traverses all dimensions except  $d$  in the downhill direction, which means in total there are at least  $r_d + 1$  arcs contributed from  $g^d$ .

Else, if  $\mathcal{C}$  does not contain cross  $g^d$ -link, then it only traverses  $g^d$ . Recall that traversing  $g^d$  is guided by the closed walk constructed in Lemma 6 and with each (FINISHED) visit to nodes there is an increment, i.e. an uphill traversal. Hence,  $g^d$  in this case contributes a number of arcs equal to the number of FINISHED visits, which in turn is the number of its nodes, or  $|V[g^d]| \geq r_d + 1$ .

Else,  $\mathcal{C}$  both traverses and crosses  $g^d$ . Then there are links with equal  $d$ th digits at their endpoints which shrink into loop links. We remove all arcs  $(\ell', \ell'') \in \mathcal{C}'$  where  $\ell'$  or  $\ell''$  is a loop link, as well as loop arcs. As a result, parts of  $\mathcal{C}'$  along which the  $d$ th digit does not change, is eliminated and  $\mathcal{C}'$  is segmented into separate PoDs. Let  $\mathcal{S} \subset \mathcal{C}'$  denote the set of remaining arcs (tails and heads of which in  $\tilde{g}^d$ ). Notice that arcs in  $\mathcal{S}$  form disconnected PoDs. Moreover, for each PoD  $P \subseteq \mathcal{S}$ , the tail of the first arc and the head of the last arc belongs to  $\tilde{g}^d$ . The remaining arcs (which do not include any  $g^d$ -link) are in  $\bar{\mathcal{S}} := \mathcal{C}' \setminus \mathcal{S}$ . Due to the segmentation of  $\mathcal{C}$ ,  $\bar{\mathcal{S}}$  forms disconnected PoDs, each beginning with an arc tailed at a link in  $\tilde{g}^d$  and ends at an arc headed at link in  $\tilde{g}^d$ . Since these PoDs cross  $g^d$  only at their end links, we apply Lemma 11 to each PoD  $P' \subseteq \bar{\mathcal{S}}$  and we obtain a PoD  $\tilde{P}$  induced by  $BP_{g^d}$ . Then, by adding each obtained  $\tilde{P}$  to  $\mathcal{C}$ , we reconnect all consecutive PoDs in  $\mathcal{S}$  and join them into a CoD  $\tilde{\mathcal{C}}$  induced by  $BP_{g^d}$ , which means  $|\tilde{\mathcal{C}}| \geq r_d + 1$ . Due to proof of Lemma





■ **Figure 1** Maximally resilient schemes for  $K_4$  and  $K_5$ . The numbers on each link are the internal nodes of the link's backup path.

11, every arc in  $\tilde{\mathcal{C}}$  is either projected from an arc in  $\mathcal{C}$  that has  $g^d$ -links as endpoints, i.e., crossing  $g^d$ , or is projected from some arc in  $\mathcal{C}$  that traverses  $g^d$  in the uphill direction. Thus by definition 9, every arc in  $\tilde{\mathcal{C}}$  is the contribution of  $g^d$ . ◀

#### 4 Generalized Hypercubes and Tori

We have described above how to construct a maximally resilient scheme for Cartesian products of given base graphs using their well-structured schemes. In this section, we showcase examples of these base graphs and apply our results to their products. In particular, we will present efficient and robust path restoration schemes for generalized hypercube graphs and tori.

##### 4.1 Complete Graphs and Generalized Hypercubes

A complete graph over  $n$  nodes is defined as  $K_n = (V, E)$  where  $V = \{0, \dots, n-1\}$  and the links  $E = \{\{i, j\} | i, j \in V, i \neq j\}$ . We present a  $(n-2)$ -resilient scheme for  $K_n$  denoted by  $BP_{K_n}$ , which we later leverage for generalized hypercubes. In the following assume every increment (+1) is performed in modulo  $n$  and it skips 0. That is,  $i+1 \equiv i \pmod{n-1} + 1$ . We generate all backup paths in two simple cases as described in Algorithm 3.

■ **Algorithm 3** Construction of  $BP_{K_n}$ .

- 
- |    |  |                                       |
|----|--|---------------------------------------|
| 1: | for each link $\ell \in E[K_n]$ do     |                                       |
| 2: | if $0 \in \ell$ then                   | ▷ i.e. $\ell = \{0, i\}$              |
| 3: | $BP_{K_n}(\ell) = [0, i+1, i]$         |                                       |
| 4: | else                                   | ▷ i.e. $\ell = \{i, j\}, i, j \neq 0$ |
| 5: | $BP_{K_n}(\ell) = [i, j+1, 0, i+1, j]$ |                                       |
- 

► **Theorem 13.** *The backup path scheme  $BP_{K_n}$  is  $(n-2)$ -resilient.*

**Proof.** The dependencies from a link  $\{i, j\}$  where  $i, j \neq 0$ , to other links can be observed in four distinct types:  $\{i, j\} \xrightarrow{A} \{i, j+1\}$ ,  $\{0, j\} \xrightarrow{B} \{0, j+1\}$ ,  $\{i, j\} \xrightarrow{C} \{0, j+1\}$  and  $\{0, j\} \xrightarrow{D} \{j, j+1\}$ . Note that with each type,  $i$  and  $j$  are interchangeable due to the symmetry of BP produced at Line 3.5. In Figure 1 (right), an exemplary CoD that consists of all the four types can be:  $\{1, 2\} \rightarrow \{1, 3\} \rightarrow \{0, 4\} \rightarrow \{0, 1\} \rightarrow \{1, 2\}$ . Next, we show that any CoD consists of at least  $n-1$  arcs, implying  $n-2$  resiliency. If  $\mathcal{C}$  consists of links all incident to some node  $i \neq 0$ , then  $\mathcal{C} = \{i, j\} \xrightarrow{A} \{i, j+1\} \xrightarrow{A} \{i, j+2\} \dots \{i, n-1\} \xrightarrow{C} \{i, 0\} \xrightarrow{D} \{i, i+1\} \xrightarrow{A} \dots \{i, j\}$ . Clearly  $\mathcal{C}$  consists of  $n-1$  arcs and therefore in the remainder we focus on CoDs over links non-incident to the same node.

Given a CoD  $\mathcal{C}$ , we construct a sequence of node ids  $S = (v_0, v_1, \dots, n-1, \dots, v_0)$  such that for every  $0 \leq t < |S|$ , it holds  $S_{t+1} \leq S_t + 1$ , and the tail of the  $t$ -th arc in  $\mathcal{C}$  is a link  $\{S_t, *\}$ . Observe that such sequence implies there are  $|S| \geq n-1$  arcs in  $\mathcal{C}$ . We construct  $S$  as follows.

1. All dependencies in  $\mathcal{C}$  are of type  $A$ . Assume the packet  $p$  that realizes the CoD is currently at node  $i$  and hits the failed link  $\{i, j\} \not\equiv 0$ . Let  $\mathcal{S}$  be the sequence of nodes that  $p$  visits until it arrives back to  $i$ . The next failure (by type  $A$ ) is either  $\{i+1, j\}$  or  $\{i, j+1\}$ . Therefore  $p$  either is rerouted to the node  $i+1$  or it stays at  $i$ . That is,  $p$  visits all nodes contiguously before it arrives back to  $i$ . After applying type  $A$  to either of the outcomes and repeating on each consequent link in a similar way, the packet visits all nodes contiguously.
2. All dependencies in  $\mathcal{C}$  are of type  $B$ . We take the sequence of non-zero endpoints. I.e.,  $\mathcal{S}[t] = v \in \mathcal{C}_t, v \neq 0$ .
3.  $\mathcal{C}$  contains multiple arc types. We refer to a path of arcs all in type  $X$  as type  $X$ -PoD. We split  $\mathcal{C}$  into maximal dependency paths of types  $A$  and  $B$ , which are concatenated by dependency arcs of type  $C$  and  $D$ . We extract a sub-sequence from each maximal PoDs and patch them into a single sequence  $\mathcal{S}$  as follows. Initially, let  $\mathcal{S} = \emptyset$  and start with a maximal  $A$ -PoD  $\{i_0, j_0\} \xrightarrow{A}, \dots$  chosen arbitrarily.
  - a. Given a  $A$ -PoD, say  $\{i, j\} \xrightarrow{A}, \dots, \xrightarrow{A} \{i', j'\}$ , the packet that realizes the PoD visits two sub-sequences depending on whether it starts at  $i$  or  $j$ . Let  $S_1$  and  $S_2$  be the produced sub-sequences ending with  $i'$  and  $j'$  respectively. The  $A$ -PoD is followed by a type  $C$  arc, that is  $\{i', j'\} \xrightarrow{C} \{i'+1, 0\}$  or  $\{i', j'\} \xrightarrow{C} \{0, j'+1\}$ . With the first case, pick the sequence  $S_1$ , otherwise pick  $S_2$ . Append to  $\mathcal{S}$  the chosen sequence and then the incremented node id at the head of the  $C$ -arc (i.e.  $i'+1$  or  $j'+1$ ).
  - b. If  $\mathcal{C}$  proceeds with a  $B$ -PoD then append to  $\mathcal{S}$  the sequence of non-zero node ids.
  - c. After the  $C$ -arc and possibly a  $B$ -PoD, there must be a  $D$ -arc. E.g.,  $\{0, j''\} \xrightarrow{D} \{j'', j''+1\}$ . The  $D$ -arc is then followed by a  $A$ -PoD (possibly the first one). If we are back to the first  $A$ -PoD, i.e.,  $\{j'', j''+1\} = \{i_0, j_0\}$ , then  $\mathcal{S}$  is already a circular sequence. Else, we continue the construction by repeating from step (a)

It is easy to see that the current sequence is contiguous after (a), (b) and (d). In particular, after (d),  $\mathcal{S}$  ends with  $j''$  and any sub-sequence chosen next in (a) begins with  $j''$  or  $j''+1$ . In either case the claim is preserved.  $\blacktriangleleft$

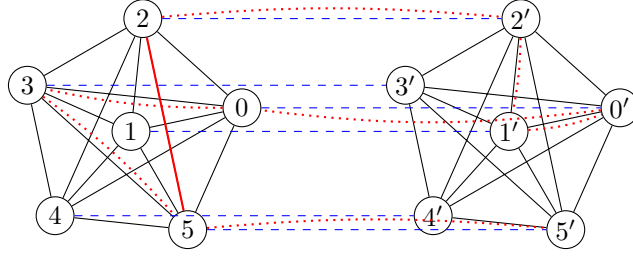
In the following lemmata, we show that this scheme is well-structured. First, we need to determine the boundary links.

► **Lemma 14.** *Every CoD induced by the scheme from Theorem 13 includes a link in  $B_{K_n} := \{\{1, i\} \mid 0 \leq i \leq n-1\}$  and the subset of arcs  $\{\{i, n-1\} \rightarrow \{i, 1\} \mid i \in \{0, 2, 3, \dots, n-2\}\} \cup \{\{1, n-1\} \rightarrow \{0, 1\}\}$  are feedback arcs.*

**Proof.** The sequence  $\mathcal{S}$  constructed in the Proof 13 contains every non-zero node id regardless of the given CoD. This means that for any node  $v \in \{1, \dots, n-1\}$ , every CoD includes some link incident to  $v$ . We pick  $v = 1$  w.l.o.g. We identify feedback arcs as those that head to a boundary link which is a unique arc in every CoD except the one induced by  $B_{K_n}$ . For this case (i.e.  $CoD(1)$ ), we designate  $\{1, n-1\} \rightarrow \{0, 1\}$  as the feedback arc.  $\blacktriangleleft$

Next, we observe the properties required by Definition 5.

► **Lemma 15.** *The scheme  $BP_{K_n}$  (Theorem 13) is well-structured.*



■ **Figure 2** A  $(6, 2)$ -cube. Each dashed blue line is a  $K_2$ -instance. They connect the two  $K_6$ -instances. They admit (respectively) 0- and 4-resilient schemes. The dotted line traces  $BP_G(\{2, 5\}) = [2, 2', 1', 0', 0, 3, 5]$ . On  $K_6$ , Lemma 6 gives the walk  $0, 1, 0, 2, 1, 3, 1, 4, 1, 5, 1$  over the boundary links of  $K_6$ , which are all the links incident to 1. The FINISHED order is  $0, 1, 2, 3, 4, 5$ . In turn, Algorithm 2 generates backup paths such as  $BP_G(\{0, 0'\}) = [0, 1, 1', 0']$  and  $BP_G(\{1, 1'\}) = [1, 0, 2, 2', 0', 1']$ . Hence,  $K_2$ -instances induce the CoD:  $\{0, 0'\} \rightarrow \{1, 1'\} \rightarrow \{2, 2'\} \rightarrow \{3, 3'\} \dots \{0, 0'\}$ . Observe in example CoDs  $\{2, 5\} \xrightarrow{*} \{2', 1'\} \rightarrow \{0', 3'\} \rightarrow \{0', 4'\} \rightarrow \{0', 5'\} \rightarrow \{1, 5\} \rightarrow \{2, 5\}$  and  $\{2, 5\} \xrightarrow{*} \{2, 2'\} \rightarrow \{2, 1\} \rightarrow \{2, 0\} \rightarrow \{2, 3\} \rightarrow \{2, 4\} \rightarrow \{2, 5\}$ , the starred arcs are counted as the contribution of  $K_2$  ( $0 + 1$  arcs), while the rest are the contribution of  $K_6$  ( $4 + 1$  arcs).

**Proof.** We observe the conditions in Definition 5 as follows. The set of boundary links in Lemma 14 form a single CoD. Moreover, for every  $v \in V[K_n], v \neq 1$ , we have  $B_{K_n}(v) = \{1, v\}$  and  $B_{K_n}(1) = \{1, 0\}$ , which means every CoD has some link in  $L_{BP_{K_n}}^*$  as the endpoint of some arcs. Therefore the procedure 5.2 can break all CoDs. Definition 5.3 can be observed in the proof of Theorem 13. ◀

Next, we formally define the generalized hypercube (GHC) as a special product graph. Given  $r_i > 0, i \in [k]$ , nodes in  $(r_k, \dots, r_1)$ -cube are represented as  $k$ -tuples  $(a_k, \dots, a_1), \forall i \in [k] : 0 \leq a_i < r_i$  (Figure 2). Therefore there are  $\prod_{i \in [k]} r_i$  nodes in a  $k$ -GHC. Every two nodes  $(a_k, \dots, a_1)$  and  $(b_k, \dots, b_1)$  that differ only at their  $i$ th digit, say  $a_i$  and  $b_i$ , are connected by an  $i$ -dim link. The degree of each node is  $\Delta = \sum_{i \in [k]} (r_i - 1)$  and the graph is  $\Delta$ -connected. Observe that  $i$ -dim links form cliques of  $r_i$  nodes. More precisely, there are  $\prod_{j \neq d} r_j$  instances of  $K_{r_d}$  for every  $1 \leq d \leq k$ . Thus, Algorithm 2 integrates individual complete graph's schemes into one scheme  $BP_{GHC}$ . See Figure 2 for an example.

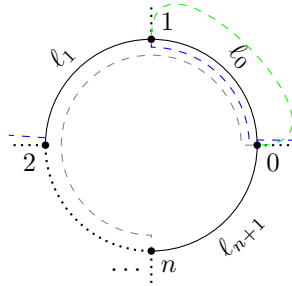
► **Corollary 16.** *The backup path scheme  $BP_{GHC}$  is  $(\Delta - 1)$ -resilient.*

**Proof.** By Lemma 15, the scheme from Theorem 13 is well-structured. Due to the fact that a GHC is the Cartesian product of complete graphs, we can apply Theorem 12 which directly implies the claim. ◀

Observe that  $\Delta$  failures can disconnect generalized hypercubes, i.e.,  $(\Delta - 1)$ -resiliency is the best we can hope for.

## 4.2 Torus and Grid

Let  $\mathcal{B} := \{C_{n_1}, \dots, C_{n_k}\}$  be a given set of base graphs where each  $C_{n_d}, d \in [k]$  is a cycle on  $n_d$  nodes. A  $k$ -dimensional torus  $\mathcal{T}$  is the Cartesian Product of  $k$  cycles. That is,  $\mathcal{T} = \prod_{d \in [k]} C_{n_d}$ . Consider a cycle  $C_n \in \mathcal{B}$  and its links  $\ell_0, \ell_1, \dots, \ell_{|n|-1}$  as they appear on the cycle. Any cycle is 1-resilient since simply every link includes every other link on its backup path:  $\forall \ell \in E[C_n] : BP_{C_n}(\ell) = E[C_n] \setminus \{\ell\}$ . Clearly,  $BP_{C_n}$  induces  $\binom{n}{2}$  CoDs, each on two arcs. The set  $B = E[C_n] \setminus \{\ell_0\}$  includes a link from every CoD, therefore it is a (minimal) set of boundary links. We choose the set of feedback arcs to be  $F := \{(\ell_i, \ell_j) \mid 0 \leq i < j \leq |n| - 1\}$ . Observe that it includes one of the two links in every min-CoD.



**Figure 3** Solid lines are links of the cycle graph  $C_{n+1}$ . Dotted lines perpendicular to the cycle represent incident links that belong to a base graph in another dimension. Dashed lines follow backup paths in  $BP_{\mathcal{G}}$  where  $\mathcal{G}$  is the Cartesian product of  $C_{n+1}$  and some other base graphs. The walk constructed in Lemma 6 is  $0, 1, 2, \dots, n-1, n, n-1, n-2, \dots, 2, 1, 0$ . By Lemma 17, in order to break all CoDs, the backup path of  $\ell_0$  (dashed green) detours every other link in  $C_{n+1}$  using the next dimension base graph. The backup path of  $\ell_1$  (dashed blue) takes  $\ell_0$ , but detours every other link. Similarly,  $\ell_2$  (not shown here) takes  $\ell_0, \ell_1$  on its backup path and detours  $\ell_3$  to  $\ell_{n+1}$ . This goes on until  $\ell_{n+1}$  which uses only links on the  $C_{n+1}$ .

**Lemma 17.** *The scheme  $BP_{C_n}$  is well-structured.*

**Proof.** Every link  $\ell_j \in E[C_n]$  has a non-feedback arc to every link  $\ell_i \in E[C_n], i < j$  (i.e.  $(\ell_j, \ell_i) \notin F$ ). Any CoD includes at least one arc  $(\ell_{j'}, \ell_{i'})$  where  $j' > i'$ . Hence it includes at least one non-feedback arc, which satisfies Definition 5 trivially. ◀

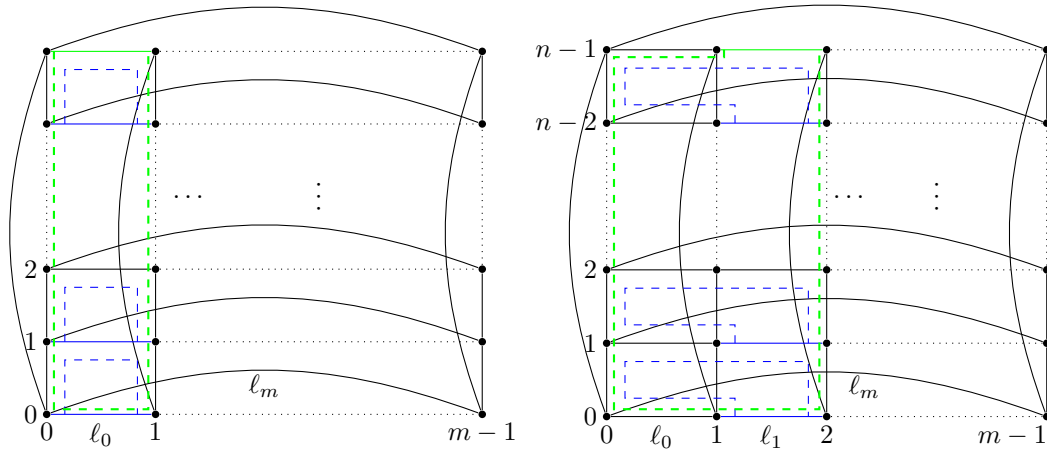
Now that we know  $BP_{C_n}$  is well-structured, we construct  $BP_{\mathcal{T}}$  using Algorithm 2 and apply Theorem 12 directly. (See Figure 3 and Figure 4 for an illustration, in the appendix)

**Corollary 18.** *The backup path scheme  $BP_{\mathcal{T}}$  is  $(2k-1)$ -resilient on the  $k$ -dimensional torus  $\mathcal{T}$ .*

As a  $k$ -dimensional torus can be disconnected by  $2k$  failures, our scheme is maximally resilient.

Next, we address  $k$ -dimensional grids via a reduction to torus. By the construction of  $BP_{\mathcal{T}}$ , only the link  $\ell_0 \in C_n$  has a feedback arc to every other link in  $C_n$ . Let  $\ell_0^d \in C_{n_d}$  be the link that corresponds to  $\ell_0$  in the base graph  $C_{n_d}$ , for every  $d \in [k]$ . Let  $\mathcal{B}' = \{P_{n_1}, \dots, P_{n_k}\}$  be the set of paths where each  $P_{n_d}$  is obtained by removing  $\ell_0^d$  from  $C_{n_d} \in \mathcal{B}$  (i.e.  $P_{n_d} = C_{n_d} \setminus \ell_0^d$ ). We construct a scheme for the grid  $\mathcal{M} = \prod_{d \in [k]} P_{n_d}$  as follows. Consider the scheme  $BP_{\mathcal{T}}$  from Corollary 18. For every  $d \in [k]$  and every backup path that uses (an instance of)  $\ell_0^d \in C_{n_d}$ , we replace  $\ell_0^d$  with its backup path. Formally,  $\forall d \in [k], \ell \in E[\mathcal{T}], \ell \neq \ell_0^d : BP_{\mathcal{M}}(\ell) = (BP_{\mathcal{T}}(\ell) \setminus \ell_0^d) \cup BP_{\mathcal{T}}(\ell_0^d)$ . Since every  $\ell \in E[\mathcal{T}], \ell \neq \ell_0^d$  includes  $\ell_0^d$  on its backup path, (after short-cutting wherever applies) we have a backup path  $BP_{\mathcal{M}}(\ell)$  for every  $\ell \in E[\mathcal{M}]$ . Each dependency to or from  $\ell_0^d, d \in [k]$  is now replaced by a dependency to a link on  $BP_{\mathcal{T}}(\ell_0^d)$ . Hence, we have replaced PoDs of two arcs with one arc, which in turn reduces the length of some min-CoDs by one. Hence, the  $(2k-1)$ -resilient scheme is reduced to a  $(2k-1-k) = (k-1)$ -resilient scheme  $BP_{\mathcal{M}}$ . As a  $k$ -dimensional grid can be disconnected by  $k$  failures, we obtain a maximally resilient scheme:

**Theorem 19.** *The backup path scheme  $BP_{\mathcal{M}}$  is  $(k-1)$ -resilient on the  $k$ -dimensional grid  $\mathcal{M}$ .*



■ **Figure 4** Each solid line is a link of the 2-dimensional  $m \times n$  torus  $\mathcal{T}$ , which is the Cartesian product of  $C_m$  and  $C_n$ . Horizontal cycles are  $C_m$ -instances and vertical cycles are  $C_n$ -instances. Dashed lines depict example backup paths in  $BP_{\mathcal{T}}$ . In the left picture, backup path of four instances of  $l_0 \in C_m$  are shown. Notice how all instances of  $l_0$  use each other sequentially on their backup paths. The backup path of  $l_0$  in the  $n$ th instance (in green, thick) has to detour all the other  $l_0$ 's in order to use the  $l_0$ -instance at row 0. This is imposed by the walk on  $C_n$  constructed in Lemma 6 (Figure 3). Also notice backup paths of  $l_1$ 's on the right picture. The only difference backup paths of  $l'_0$ 's is that they use the  $l_0$  in the same instance before proceeding to the next  $C_m$ -instance. In a similar fashion, each  $l_2$ -instance uses  $l_0, l_1$  in the same  $C_m$ -instance and so on, up to  $l_m$  which uses only the links on the same  $C_m$ -instance.

## 5 Conclusion and Future Work

This paper studied the design of algorithms for local fast failover in the setting that requires guaranteed (policy and function preserving) visits to every waypoint along the original path, under multiple link failures. Our main result is a maximally resilient backup path scheme for the Cartesian product of any set of base graphs, as long as for each base graph a well-structured scheme is provisioned. We showcased applications of this result using complete graphs, cycles, and paths by providing a well-structured scheme for each base graph separately. This allowed us to devise algorithms for important network topologies, such as generalized hypercubes and tori. In general, the result applies to the product of any combination of these base graphs as well.

We see our work as a first step and believe that it opens several promising directions for future research. From a dependability perspective, the main open question is whether  $k$ -connectivity is always sufficient for  $(k - 1)$ -resiliency w.r.t. backup paths. It might be insightful to understand the logic behind schemes formulated by Definition 5.

---

## References

- 1 Saeed Akhoondian Amiri, Klaus-Tycho Foerster, Riko Jacob, and Stefan Schmid. Charting the algorithmic complexity of waypoint routing. *Comput. Commun. Rev.*, 48(1):42–48, 2018.
- 2 Alia K Atlas and Alex Zinin. Basic specification for IP fast-reroute: loop-free alternates. *IETF RFC 5286*, 2008.
- 3 Michael Borokhovich and Stefan Schmid. How (not) to shoot in your foot with SDN local fast failover: A load-connectivity tradeoff. In *Proc. OPODIS*, 2013.

- 4 B. Carpenter and S. Brim. Middleboxes: Taxonomy and issues. RFC 3234, RFC Editor, February 2002. URL: <http://www.rfc-editor.org/rfc/rfc3234.txt>.
- 5 Marco Chiesa, Andrei V. Gurtov, Aleksander Madry, Slobodan Mitrovic, Ilya Nikolaevskiy, Michael Schapira, and Scott Shenker. On the resiliency of randomized routing against multiple edge failures. In *Proc. ICALP*, 2016.
- 6 Marco Chiesa, Andrzej Kamisiński, Jacek Rak, Gábor Rétvári, and Stefan Schmid. A Survey of Fast Recovery Mechanisms in the Data Plane. *TechRxiv*, 2020. URL: [https://www.techrxiv.org/articles/preprint/Fast\\_Recovery\\_Mechanisms\\_in\\_the\\_Data\\_Plane/12367508](https://www.techrxiv.org/articles/preprint/Fast_Recovery_Mechanisms_in_the_Data_Plane/12367508).
- 7 Marco Chiesa, Ilya Nikolaevskiy, Slobodan Mitrovic, Andrei V. Gurtov, Aleksander Madry, Michael Schapira, and Scott Shenker. On the resiliency of static forwarding tables. *IEEE/ACM Trans. Netw.*, 25(2):1133–1146, 2017.
- 8 Marco Chiesa, Ilya Nikolaevskiy, Slobodan Mitrovic, Aurojit Panda, Andrei V. Gurtov, Aleksander Madry, Michael Schapira, and Scott Shenker. The quest for resilient (static) forwarding tables. In *Proc. IEEE INFOCOM*, 2016.
- 9 Hongsik Choi, Suresh Subramaniam, and Hyeong-Ah Choi. On double-link failure recovery in WDM optical networks. In *Proc. IEEE INFOCOM*, 2002.
- 10 Theodore Elhourani, Abishek Gopalan, and Srinivasan Ramasubramanian. IP fast rerouting for multi-link failures. *IEEE/ACM Trans. Netw.*, 24(5):3014–3025, 2016.
- 11 ETSI. Network functions virtualisation. In *White Paper*, 2013.
- 12 Joan Feigenbaum, Brighten Godfrey, Aurojit Panda, Michael Schapira, Scott Shenker, and Ankit Singla. Brief announcement: on the resilience of routing tables. In *Proc. ACM PODC*, 2012.
- 13 Klaus-Tycho Foerster, Juho Hirvonen, Yvonne-Anne Pignolet, Stefan Schmid, and Gilles Trédan. On the feasibility of perfect resilience with local fast failover. In *Proc. APOCS*, 2021.
- 14 Klaus-Tycho Foerster, Andrzej Kamisiński, Yvonne-Anne Pignolet, Stefan Schmid, and Gilles Trédan. Bonsai: Efficient fast failover routing using small arborescences. In *Proc. IEEE/IFIP DSN*, 2019.
- 15 Klaus-Tycho Foerster, Andrzej Kamisiński, Yvonne-Anne Pignolet, Stefan Schmid, and Gilles Trédan. Improved fast rerouting using postprocessing. In *Proc. IEEE SRDS*, 2019.
- 16 Klaus-Tycho Foerster, Mahmoud Parham, Marco Chiesa, and Stefan Schmid. TI-MFA: keep calm and reroute segments fast. In *Global Internet Symposium (GI)*, 2018.
- 17 Klaus-Tycho Foerster, Mahmoud Parham, Stefan Schmid, and Tao Wen. Local fast segment rerouting on hypercubes. In *Proc. OPODIS*, 2018.
- 18 Klaus-Tycho Foerster, Yvonne-Anne Pignolet, Stefan Schmid, and Gilles Trédan. Local fast failover routing with low stretch. *Comput. Commun. Rev.*, 48(1):35–41, 2018.
- 19 Klaus-Tycho Foerster, Yvonne-Anne Pignolet, Stefan Schmid, and Gilles Trédan. Casa: Congestion and stretch aware static fast rerouting. In *Proc. IEEE INFOCOM*, 2019.
- 20 Pierre François, Clarence Filsfils, Ahmed Bashandy, and Bruno Decraene. Topology Independent Fast Reroute using Segment Routing. Internet-Draft draft-francois-segment-routing-ti-lfa-00, Internet Engineering Task Force, November 2013. URL: <https://datatracker.ietf.org/doc/html/draft-francois-segment-routing-ti-lfa-00>.
- 21 Pierre François, Clarence Filsfils, John Evans, and Olivier Bonaventure. Achieving sub-second IGP convergence in large IP networks. *Comput. Commun. Rev.*, 35(3):35–44, 2005.
- 22 Karthik Lakshminarayanan, Matthew Caesar, Murali Rangan, Tom Anderson, Scott Shenker, and Ion Stoica. Achieving convergence-free routing using failure-carrying packets. In *Proc. ACM SIGCOMM*, 2007.
- 23 Eunseuk Oh, Hongsik Choi, and Jong-Seok Kim. Double-link failure recovery in WDM optical torus networks. In *Proc. ICOIN*, 2004.
- 24 P. Pan, G. Swallow, and A. Atlas. Fast reroute extensions to RSVP-TE for LSP tunnels. RFC 4090, RFC Editor, May 2005.
- 25 Yvonne-Anne Pignolet, Stefan Schmid, and Gilles Trédan. Load-optimal local fast rerouting for resilient networks. In *Proc. IEEE/IFIP DSN*, 2017.

## 26:16 Maximally Resilient Replacement Paths for a Family of Product Graphs

- 26 Stefan Schmid and Jiri Srba. Polynomial-time what-if analysis for prefix-manipulating mpls networks. In *Proc. IEEE INFOCOM*, 2018.
- 27 Brent Stephens, Alan L. Cox, and Scott Rixner. Plinko: Building provably resilient forwarding tables. In *Proc. ACM HotNets*, 2013.
- 28 Brent Stephens, Alan L Cox, and Scott Rixner. Scalable multi-failure fast failover via forwarding table compression. *SOSR. ACM*, 2016.
- 29 Baohua Yang, Junda Liu, Scott Shenker, Jun Li, and Kai Zheng. Keep forwarding: Towards k-link failure resilient routing. In *Proc. IEEE INFOCOM*, 2014.



# Self-Stabilizing Byzantine-Resilient Communication in Dynamic Networks

Alexandre Maurer

Mohammed VI Polytechnic University, School of Computer Science, Ben Guerir, Morocco  
alexandre.maurer@um6p.ma

---

## Abstract

We consider the problem of communicating reliably in a dynamic network in the presence of up to  $k$  Byzantine failures. It was shown that this problem can be solved if and only if the dynamic graph satisfies a certain condition, that we call “RDC condition”. In this paper, we present the first self-stabilizing algorithm for reliable communication in this setting – that is: in addition to permanent Byzantine failures, there can also be an arbitrary number of transient failures. We prove the correctness of this algorithm, provided that the RDC condition is “always eventually satisfied”.

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

**Keywords and phrases** Dynamic networks, Self-stabilization, Byzantine failures

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2020.27

## 1 Introduction

As networks grow larger and larger, it becomes more and more likely that some of their nodes will behave incorrectly at some point, for various reasons. Therefore, it is crucial to design *robust* networks, that is: networks that still satisfy some essential properties even when some nodes fail. There are many models of node failure, but the most general one is the Byzantine model [13]: we assume that the failing nodes can have any arbitrary behavior. Thus, we encompass any possible type of failure.

Here, we consider the problem of *reliable communication*: any two correct nodes of the network should be able to exchange messages reliably, despite the potentially malicious behavior of some Byzantine nodes.

One way to solve this problem is to use cryptography [6, 10]: the nodes use digital signatures to authenticate the sender across multiple hops. However, cryptography is not always reliable (see, for instance, the Heartbleed bug [1] discovered in the widely deployed OpenSSL software). The Defense in Depth paradigm [14] recommends the use of multiple security layers, including non-cryptographic layers. For instance, if the cryptographic layer is compromised (bug, virus, . . .), a cryptography-free communication layer can be used to safely broadcast a patch, or to update cryptographic keys. Thus, even when cryptography is available, it is interesting to develop non-cryptographic solutions. In the following, we focus on these non-cryptographic solutions.

When it comes to non-cryptographic solutions, many solutions have been proposed for *static* networks (e.g. [4, 11, 22, 15, 17, 16, 7, 21, 8]). Fewer papers consider *dynamic* networks (e.g. [20, 3, 19]), where the topology changes over time. In particular, [19] showed the necessary and sufficient condition on the graph topology to tolerate up to  $k$  Byzantine nodes in a dynamic network. We call this condition the **RDC (Reliable Dynamic Communication)** condition. (This condition is described in Section 2.3, but its exact terms do not matter at this point.)

There already exists an algorithm for reliable communication in dynamic networks tolerating permanent Byzantine failures [19]. Hence, the question: is it possible to go further in terms of reliability guarantees?



© Alexandre Maurer;

licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 27; pp. 27:1–27:11

Leibniz International Proceedings in Informatics



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

The concept of *multitolerance* [2, 12] describes the property of a system to tolerate multiple fault-classes. One of the strongest possible level of multitolerance is the following: tolerating, not only a given number of permanent Byzantine failures, but also an *arbitrary* number of transient failures. This second point consists in assuming that the system can have any arbitrary initial state. More precisely: (1) each correct node can have any arbitrary initial state, and (2) communication channels between nodes can contain arbitrary messages, “sent but not yet received”. In other words, such an algorithm is *self-stabilizing* [9]: it can recover from any incorrect initial state. Satisfying this property in the presence of Byzantine failures can be particularly challenging: Byzantine nodes, in addition to their usual malicious behavior, can also actively try to prevent stabilization. The problem of reliable communication despite transient and Byzantine failures was already considered in [18], but for a specific class of static networks and a non-standard criteria on Byzantine failures (i.e. the distance between Byzantine failures).

**Our contribution is the following.** we present the first *self-stabilizing* Byzantine-resilient algorithm for reliable communication in a dynamic network. We prove the correctness of our algorithm under the following assumption: the aforementioned RDC condition is “always eventually satisfied” – that is: for any time  $t$ , there always exists a time  $t' \geq t$  where the RDC condition is satisfied.

**The rest of the paper is organized as follows.** In Section 2, we present the setting (definitions, assumptions...). In Section 3, we describe the problem. In Section 4, we motivate the main assumption w.r.t. the problem (i.e., that the RDC condition is “always eventually satisfied”). In Section 5, we describe our algorithm. In Section 6, we prove its correctness.

## 2 Preliminaries

The setting is mostly similar to [19]. We recall several definitions below.

### 2.1 Network model

We consider a continuous temporal domain  $\mathbb{R}^+$ . We model the system as a time varying graph, as defined by Casteigts, Flocchini, Quattrociochi and Santoro [5], where vertices represent the processes and edges represent the communication links (or channels). A time varying graph is a dynamic graph represented by a tuple  $\mathcal{G} = (V, E, \rho, \zeta)$  where:

- $V$  is the set of *nodes*.
- $E \subseteq V \times V$  is the set of *edges*.
- $\rho : E \times \mathbb{R}^+ \rightarrow \{0, 1\}$  is the *presence* function:  $\rho(e, t) = 1$  indicates that edge  $e$  is present at time  $t$ .
- $\zeta : E \times \mathbb{R}^+ \rightarrow \mathbb{R}^+$  is the *latency* function:  $\zeta(e, t) = T$  indicates that a message sent at time  $t$  through edge  $e$  will be received, at worst, at time  $t + T$ .

(Note: although one could imagine a simpler definition, we prefer to stick with the standard definition of [5].)

### 2.2 Definitions

Informally, a *dynamic path* is a sequence of nodes a message can traverse, with respect to network dynamicity and latency.

► **Definition 1** (Dynamic path). A sequence of distinct nodes  $(u_1, \dots, u_n)$  is a “dynamic path from  $u_1$  to  $u_n$  starting after  $t_0$ ” if there exists a sequence of times  $(t_1, \dots, t_n)$  such that  $t_0 \leq t_1 \leq t_2 \leq \dots \leq t_n$ , and,  $\forall i \in \{1, \dots, n-1\}$ , we have:

- $e_i = (u_i, u_{i+1}) \in E$ , i.e., there exists an edge connecting  $u_i$  to  $u_{i+1}$ .
- $\forall t \in [t_i, t_i + \zeta(e_i, t_i)]$ ,  $\rho(e_i, t) = 1$ , i.e.,  $u_i$  can send a message to  $u_{i+1}$  at time  $t_i$ .
- $\zeta(e_i, t_i) \leq t_{i+1} - t_i$ , i.e., the aforementioned message is received by time  $t_{i+1}$ .

With this definition, we can now define the following elements:

- Let  $Dyn(p, q, t_0)$  be the set of node sets  $\{u_1, \dots, u_n\}$  such that  $(p, u_1, \dots, u_n, q)$  is a dynamic path starting after  $t_0$  (if  $(p, q)$  is a dynamic path starting after  $t_0$ ,  $Dyn(p, q, t_0)$  contains an empty set).
- For any nonempty set of nonempty node sets  $X = \{S_1, \dots, S_n\}$ , let  $Cut(X)$  be the set of node sets  $C$  such that,  $\forall i \in \{1, \dots, n\}$ ,  $C \cap S_i \neq \emptyset$  ( $C$  contains at least one node from each set  $S_i$ ).<sup>1</sup>
- For any nonempty set of node sets  $X$ , we define  $MinCut(X)$  as follows:
  - If  $X$  contains an empty set,  $MinCut(X) = +\infty$ .<sup>2</sup>
  - Otherwise:  $MinCut(X) = \min_{C \in Cut(X)} |C|$  (the size of the smallest element of  $Cut(X)$ ).

We say that a node *multicasts* a message  $m$  when it sends  $m$  to all nodes in its current local topology.<sup>3</sup>

## 2.3 Setting and assumptions

We make the same basic assumptions as previous works on the subject (e.g. [4, 7, 11, 15, 16, 17, 21, 22]):

1. Each node has a unique identifier.
2. When a node  $q$  receives a message through channel  $(p, q)$ , it knows that  $p$  sent the message.
3. Each node  $u$  is aware of its *local topology* at any given time  $t$  (here,  $u$  knows the set of nodes  $v$  such that  $\rho((u, v), t) = 1$ )<sup>4</sup>.
4. The time required for computation and sending messages is negligible w.r.t. the delays between changes in the dynamic graph.

**Permanent Byzantine failures.** An omniscient adversary can select up to  $k$  nodes as *Byzantine*. These nodes can have a totally arbitrary and unpredictable behavior defined by the adversary (including tampering or dropping messages, or simply crashing). Of course, correct nodes are unable to know *a priori* which nodes are Byzantine.

<sup>1</sup> Here, “one node” can be any node of  $S_i$ .

<sup>2</sup> Here, an empty set corresponds to the case where the two nodes  $p$  and  $q$  trying to communicate are directly connected at some point. Therefore, no amount of (other) nodes suppression will be sufficient to disconnect them.

<sup>3</sup> We use “multicast” instead of “broadcast” here to avoid common misunderstandings: “broadcast” can refer to some very specific problems in distributed computing (e.g. “Byzantine Broadcast”), not necessarily related to our problem.

<sup>4</sup> Note that this assumption is necessary to ensure that each existing dynamic path can be explored. For instance, it is possible that some edges appear during exactly the time required to send a message. In such a situation, the sending node must be immediately aware of the topology change.

**Transient failures.** In addition to Byzantine failures, for the correct nodes, any variable of their algorithm can have any arbitrary initial value. Besides, for any two neighbor nodes  $u$  and  $v$ , a channel connecting  $u$  and  $v$  can initially contain any set of messages “sent by  $u$  but not yet received by  $v$ ”. In the following, this set is called  $Sen(u, v)$ .<sup>5</sup>

**RDC condition.** In [19], it was shown that the necessary and sufficient condition for reliable communication in a dynamic network is the following: for each pair of nodes  $\{p, q\}$ ,  $MinCut(Dyn(p, q, 0)) > 2k$ .

We call this condition the **RDC (Reliable Dynamic Communication)** condition.

**Main assumption.** In this paper, we assume that the RDC condition is “always eventually satisfied”, that is: for any two nodes  $p$  and  $q$  and for any time  $t_0$ ,  $MinCut(Dyn(p, q, t_0)) > 2k$ .<sup>6</sup> In Section 4, we motivate the “always” of “always eventually satisfied” w.r.t the problem.

### 3 The Problem

Let us assume that each correct node  $p$  has a *fixed* attribute  $p.m_0$  (the message that  $p$  wants to broadcast, not part of the initial state) and a memory set  $p.Acc$  (where received messages are stored). For each correct node  $q$ , when the set  $q.Acc$  contains a tuple  $(p, m)$ , we say that  $q$  *accepts* the message  $m$  as being from  $p$ .

**The problem we try to solve is the following:** finding an algorithm (for correct nodes) such that, given the aforementioned setting, we have the two following properties:

For any two correct nodes  $p$  and  $q$ , there exists a time  $t$  such that, after  $t$ ...

1. **[Safety]** There exists no  $m' \neq p.m_0$  such that  $(p, m') \in q.Acc$ .
2. **[Liveness]**  $(p, p.m_0) \in q.Acc$ .

The first property ensures that, after  $t$ , no wrong message (i.e., a message  $m'$  pretending to be the message  $p.m_0$  from  $p$ ) is accepted by  $q$ . The second property ensures that, after  $t$ , the correct message  $p.m_0$  is accepted by  $q$ .

Such an algorithm would be *self-stabilizing* w.r.t. these two properties: no matter what the initial state is, these properties are always eventually satisfied.<sup>7</sup>

In the following, we motivate the main assumption w.r.t this problem (Section 4), then present our algorithm (Section 5) and prove that it solves the problem (Section 6).

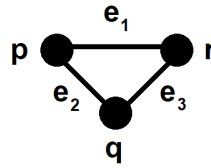
### 4 Motivation of the main assumption

In this section, we motivate the main assumption (stated in Section 2.3). In particular, we motivate the “always” in “always eventually satisfied”. In Theorem 2 below, we show that, if the RDC condition in just “eventually satisfied”, no algorithm can solve the desired problem in the setting we consider. We do this by constructing two possible scenarios (allowed in this setting) that lead to incompatible outcomes.

<sup>5</sup> We do not assume any bound on the capacity of communication channels: the number of fake initial messages “sent but not yet received” can be arbitrarily high.

<sup>6</sup> If we take “always eventually satisfied” strictly, the proposition is: for any two nodes  $p$  and  $q$  and for any time  $t$ , there exists  $t_0 \geq t$  such that  $MinCut(Dyn(p, q, t_0)) > 2k$ . But this proposition immediately implies the aforementioned (simpler) proposition.

<sup>7</sup> As usually done in self-stabilization papers, we assume that the algorithm itself is “hardwired”, and cannot be corrupted (otherwise, it would be impossible to give any guarantee: the behavior of all nodes would be completely arbitrary). However, any variable used by the algorithm can have any arbitrary initial value.



■ **Figure 1** Graph used for the proof of Theorem 2.

► **Theorem 2.** *Let us assume the aforementioned setting (with transient failures, etc), but with one small change: the RDC condition is just eventually satisfied (and not “always eventually satisfied”). Then, there exists no algorithm solving the problem of Section 3.*

**Proof.** Suppose the opposite: there exists such an algorithm. Let  $m_1$  and  $m_2$  be two distinct messages.

Let us assume that  $k = 0$  (no Byzantine failures). Then, the RDC condition simplifies as follows: there must exist one dynamic path between the sender and the receiver.

Consider the static network of Figure 1. In the following, we use this static network to describe two dynamic networks, and show a contradiction. Let  $x$  be any message that can be sent by  $q$  to  $r$ .

We first describe a scenario  $S_1(x)$  (of which  $x$  is a parameter). In this scenario,  $p.m_0 = m_1$ . The evolution of the dynamic graph is the following:

**Step 1:** Edge  $e_1$  appears, then disappears.

**Step 2:** Edge  $e_3$  appears, then disappears. During this time,  $r$  receives  $x$  from  $q$ .<sup>8</sup>

With Step 1, there exists a dynamic path from  $p$  to  $r$ . Thus, as the algorithm solves the desired problem, there exists a time  $t_1$  after which we have  $(p, m_1) \in r.Acc$ .<sup>9</sup> Let  $y$  be the state of  $r$  between Step 1 and Step 2.

We now describe a second scenario  $S_2$ . In this scenario,  $p.m_0 = m_2$ , and the initial state of  $r$  is  $y$ . The evolution of the dynamic graph is the following:

**Step 1':** Edge  $e_2$  appears, then disappears.

**Step 2':** Edge  $e_3$  appears, then disappears.

With these two steps, there exists a dynamic path from  $p$  to  $r$ . Thus, as the algorithm solves the desired problem, there exists a time  $t_2$  after which we have  $(p, m_2) \in r.Acc$ . Let  $t_3 = \max(t_1, t_2)$ , and let  $x'$  be the message sent from  $q$  to  $r$  during Step 2'.

Now, let us consider (1)  $S_1(x')$  after Step 1 and (2)  $S_2$ . From the point of view of  $r$ , what happens then is exactly the same:  $r$  starts in state  $y$ , then receives  $x'$  from  $q$ . Thus, according to both scenarios, after  $t_3$ , we have *both*  $(p, m_1) \in r.Acc$  and  $(p, m_2) \in r.Acc$ .

As the algorithm solves the desired problem, it implies that  $p.m_0 = m_1 = m_2$ , which contradicts our initial assumption ( $m_1 \neq m_2$ ). Thus, the result. ◀

## 5 Algorithm

In this section, we provide:

- An explanation of the main intuition behind the algorithm (5.1).
- A full description of our algorithm (5.2).
- A more detailed explanation of the algorithm, to facilitate its understanding (5.3).

<sup>8</sup> This can happen for any  $x$ , as the initial content of communication channels between nodes is arbitrary, according to our model (see Section 2.3).

<sup>9</sup> The fact that  $p$  sends a message to  $r$  through edge  $e_1$  is implicit here, as the outcome of the algorithm is assumed to be guaranteed by the RDC condition (in the context of this proof by contradiction).

## 5.1 Intuition behind the algorithm

The initial idea of the algorithm is similar to [19]:

- Broadcast each message through each possible dynamic path, and register the identifier of the nodes forwarding the message.
- Before accepting the message: check if the various instances of the same message satisfy the RDC condition.

This is done through rules 2, 3 and 4 of the algorithm below. However, in a setting with transient failures, this is not sufficient: for instance, it is possible that a false message has already been accepted, or that enough messages have been sent to have a false message accepted in the future.

To solve this problem, we associate an integer  $\alpha$  to each message. Now, each message looks like this:  $(s, m, S, \alpha)$ , where  $s$  is the sender (or pretending to be so),  $m$  is the content of the message, and  $S$  is the set of nodes crossed by the message.

The key idea here is the following: the algorithm is designed so that false transient messages are “stuck” with the value  $\alpha$  they initially have (as shown in Lemma 3). Therefore, if correct nodes keep broadcasting their message with increasing values of  $\alpha$  (Rule 1), and if we give priority to messages with the highest value of  $\alpha$  (Rule 5), the correct messages are eventually accepted. Of course, Byzantine nodes can still broadcast messages with arbitrarily high values of  $\alpha$ , but rules 2, 3 and 4 ensure that this will never be sufficient (according to the assumptions on the topology of the network).

## 5.2 Full description

For each correct node  $u$ , let  $u.m_0$  be the message that  $u$  wants to broadcast;  $u$  also maintains the following variables:

- An integer  $u.\alpha$  (with an arbitrary initial value).
- Three memory sets  $u.\Omega$ ,  $u.Acc_0$  and  $u.Acc$  (the initial content of these sets is completely arbitrary).<sup>10</sup>

Let  $A(u, s, m)$  be the set of integers  $\alpha$  such that  $(s, m, \alpha) \in u.Acc_0$ . Let  $Count(u, s, m) = |A(u, s, m)|$ .

In **Rule 1** below, “keep doing X” means that the algorithm will always eventually do X (i.e., if  $t$  is the current time, there always exists a time  $t' \geq t$  at which X is done).

Each correct node  $u$  obeys to the following rules:

- **Rule 1.** Keep doing the following:  $u.\alpha := u.\alpha + 1$ , and add  $\{(u, u.m_0, \emptyset, u.\alpha)\}$  to  $u.\Omega$ .
- **Rule 2.** Whenever  $u.\Omega$  or the set of neighbors of  $u$  changes<sup>11</sup>: multicast  $u.\Omega$ .
- **Rule 3.** When  $u$  receives a set  $\Omega'$  from a neighbor  $v$ :  $\forall (s, m, S, \alpha) \in \Omega'$ , if  $v \notin S$ , add  $(s, m, S \cup \{v\}, \alpha)$  to  $u.\Omega$ .
- **Rule 4.** When there exist  $s, m, \alpha$  and  $n$  sets  $S_1, \dots, S_n$  such that the following 3 conditions are satisfied:
  1.  $\forall i \in \{1, \dots, n\}, (s, m, S_i \cup \{s\}, \alpha) \in u.\Omega$ .
  2.  $MinCut(\{S_1, \dots, S_n\}) > k$ .
  3.  $(s, m, \alpha) \notin u.Acc_0$ .
 Add  $(s, m, \alpha)$  to  $u.Acc_0$ .

<sup>10</sup> See 5.3 for an explanation of the role of these memory sets.

<sup>11</sup> See assumption 3 in Section 2.3.

- **Rule 5.** When there exist  $s$  and  $m$  such that the following 3 conditions are satisfied:
  1.  $Count(u, s, m) \geq 1$ .
  2. There exists no  $m' \neq m$  such that  $Count(u, s, m) \leq Count(u, s, m')$ .
  3.  $(s, m) \notin u.Acc$ .
 Do the following:
  1.  $\forall m'$  such that  $(s, m') \in u.Acc$ , remove  $(s, m')$  from  $u.Acc$ .
  2. Add  $(s, m)$  to  $u.Acc$ .

### 5.3 Detailed explanation

In addition to the full description of the algorithm, we provide more detailed explanations here, to facilitate its understanding.

First, let us explain the role of the variables of each correct node  $u$ .

- The integer  $u.\alpha$  is a counter that can only increase. This mechanism is used to defeat false transient messages, as we explain below.
- The memory set  $u.\Omega$  stores all messages received by  $u$ , without discrimination, as described in Rule 3. The elements stored in this set are tuples  $(p, m, S, \alpha)$ , where...
  - $p$  is supposedly the author of the message;
  - $m$  is the content of the message supposedly sent by  $p$  (it can be any piece of information);
  - $S$  is the set of nodes that supposedly forwarded the message;
  - $\alpha$  is an integer associated to the rest of the tuple.
- The role of the memory set  $u.Acc_0$  is to store messages that are “pre-accepted”, that is: messages that have been (or appear to have been) sent through several dynamic paths, in accordance with the RDC condition. These messages are added to  $u.Acc_0$  in Rule 4.
- The role of the memory set  $u.Acc$  is to store messages that are “truly accepted”, that is: messages from  $u.Acc_0$  satisfying a particular condition w.r.t their integers  $\alpha$ . The goal is that, eventually,  $u.Acc$  contains each message from each correct node, and no false message (that is: a tuple  $(s, m')$  such that  $m' \neq s.m_0$ ).

Now, let us provide an informal explanation for each rule, for a given node  $u$ .

- **Rule 1:** This rule ensures that  $u$  always eventually does the following: increment its counter  $u.\alpha$ , and add its message  $u.m_0$  (associated with the new value of  $u.\alpha$ ) to its set  $u.\Omega$ .
- **Rule 2:** This rule sends the content of  $u.\Omega$  to all neighbors of  $u$  whenever  $u.\Omega$  is updated (which happens either in Rule 1 or 3).
- **Rule 3:** Whenever  $u$  receives a set  $\Omega'$  from a neighbor node  $v$ , it adds each tuple contained in  $\Omega'$  to its own set  $u.\Omega$ . However, before doing so, it adds  $v$  to the third element of the tuple (i.e., the set  $S$  supposed to register all the nodes that forwarded this tuple).
- **Rule 4:** This rule adds some elements to  $u.Acc_0$  (the set of “pre-accepted” messages) when some conditions are satisfied in  $u.\Omega$ . More precisely: there must exist a node identifier  $s$ , a message  $m$ , a value  $\alpha$  and  $n$  sets of nodes  $(S_1, \dots, S_n)$  such that...
  - For each of these node sets,  $u.\Omega$  contains a tuple associating  $s, m, \alpha$  and this node set (enriched with  $s$ ). This condition ensures that at least one of these sets actually corresponds to the correct nodes that actually forwarded the message (as shown in the correctness proof).
  - The  $n$  sets of nodes cannot be (all) cut by removing  $k$  nodes ( $MinCut(\{S_1, \dots, S_n\}) > k$ ).
  - $u.Acc_0$  does not already contain  $(s, m, \alpha)$ .



The underlying idea of Rule 4 is to prevent  $k$  Byzantine nodes from cooperating to make  $u$  add false messages to  $u.Acc_0$ . Indeed, as shown in the correctness proof,  $k$  Byzantine nodes are not sufficient to bypass the filter of Rule 4.

- **Rule 5:** This rule adds some elements to  $u.Acc$  (the set of “truly accepted” messages) when some conditions are satisfied in  $u.Acc_0$ . More precisely: when there exists a node identifier  $s$  and a message  $m$  such that the number of tuples  $(s, m, \alpha) \in u.Acc_0$  is unmatched by any other message  $m'$ ,  $(s, m)$  is added to  $u.Acc$ , and replaces any previous message  $(s, m')$ .

The underlying idea of Rule 5 is to (eventually) eliminate and replace any false transient messages that  $u.Acc_0$  may initially contain: as correct messages keep being generated with new values of  $\alpha$  (according to Rule 1), they will eventually outnumber false messages.

## 6 Correctness proof

We prove the correctness of the algorithm in Theorem 9.

Before going further, we define  $Z$ , the set of values  $\alpha$  such that a tuple  $(s, m, S, \alpha)$  exists somewhere in the system at  $t = 0$ :

- Let  $Z_1$  be the set of integers  $\alpha$  such that,  $\forall \alpha \in Z_1$ , the following proposition is true: at  $t = 0$ , there exist  $s, m, S, u, v$  and  $\Omega$  such that  $\Omega \in Sen(u, v)$  and  $(s, m, S, \alpha) \in \Omega$ .
- Let  $Z_2$  be the set of integers  $\alpha$  such that,  $\forall \alpha \in Z_2$ , the following proposition is true: at  $t = 0$ , there exists  $s, m, S$ , and  $u$  such that  $(s, m, S, \alpha) \in u.\Omega$ .
- Let  $Z = Z_1 \cup Z_2$ .

### Overview of the proof

- In Lemma 3, we show that no false message can be accepted when it is associated with an integer  $\alpha \notin Z$ . Thus, the number of integers  $\alpha$  used by false messages is bounded by  $|Z|$  (Lemma 4).
- In Lemma 5, we show that, for any  $\alpha_0$ , a correct node eventually broadcasts messages with  $\alpha \geq \alpha_0$ .
- In Lemma 6, we show that correct messages successfully broadcast along each dynamic path. Thus, as shown in Lemma 7, each correct message with a large enough  $\alpha$  is eventually “pre-accepted” (that is, added to the set  $Acc_0$  of the receiver).
- From Lemmas 5 and 7, it follows that each correct message is eventually pre-accepted (Lemma 8).
- As the number of integers  $\alpha$  used by false messages is bounded (Lemma 4), it follows that, eventually, the only messages accepted are the correct ones (Theorem 9).

► **Lemma 3.** *Let  $\alpha \notin Z$ . Let  $p$  and  $q$  be two correct nodes. Let  $m' \neq p.m_0$ . Then, we never have  $(p, m', \alpha) \in q.Acc_0$ .*

**Proof.** Suppose the opposite:  $(p, m', \alpha) \in q.Acc_0$ . Then, there exists  $\{S_1, \dots, S_n\}$  such that,  $\forall i \in \{1, \dots, n\}$ ,  $(p, m', S_i \cup \{p\}, \alpha) \in q.\Omega$  and  $MinCut(\{S_1, \dots, S_n\}) > k$ .

Suppose that each node set  $S \in \{S_1, \dots, S_n\}$  contains at least one Byzantine node. If  $C$  is the set of Byzantine nodes, then  $C \in Cut(\{S_1, \dots, S_n\})$  and  $|C| \leq k$ . This is impossible because  $MinCut(\{S_1, \dots, S_n\}) > k$ . Therefore, there exists  $S \in \{S_1, \dots, S_n\}$  such that  $S$  does not contain any Byzantine node.

Now, let us use the correct dynamic path corresponding to  $S$  to show that  $m' = p.m_0$ . Let  $n' = |S \cup \{p\}|$ . Let us show the following property  $\mathcal{P}_i$  by induction,  $\forall i \in \{0, \dots, n'\}$ : there exists a correct node  $u_i$  and a set of correct nodes  $X_i$  such that  $(p, m', X_i, \alpha) \in u_i.\Omega$  and  $|X_i| = |S \cup \{p\}| - i$ .

- As  $S \in \{S_1, \dots, S_n\}$ ,  $(p, m', S \cup \{p\}, \alpha) \in q.\Omega$ . Thus,  $\mathcal{P}_0$  is true if we take  $u_0 = q$  and  $X_0 = S \cup \{p\}$ .
- Let us now suppose that  $\mathcal{P}_i$  is true, for  $i < n'$ . As  $(p, m', X_i, \alpha) \in u_i.\Omega$ , according to Rule 3 of our algorithm, it implies that  $u_i$  received  $\Omega'$  from a node  $v$ , with  $(p, m', X, \alpha) \in \Omega'$ ,  $v \notin X$  and  $X_i = X \cup \{v\}$ . Thus,  $|X| = |X_i| - 1 = |S \cup \{p\}| - (i + 1)$ .  
As  $v \in X_i$  and  $X_i$  is a set of correct nodes,  $v$  is correct and behaves according to our algorithm. Then, as  $v$  sent  $\Omega'$ , according to Rule 2 of our algorithm, we necessarily have  $\Omega' \subseteq v.\Omega$ . Thus, as  $(p, m', X, \alpha) \in \Omega'$ , we have  $(p, m', X, \alpha) \in v.\Omega$ . Hence,  $\mathcal{P}_{i+1}$  is true if we take  $u_{i+1} = v$  and  $X_{i+1} = X$ .

By the induction principle,  $\mathcal{P}_{n'}$  is true. As  $|X_{n'}| = 0$ ,  $X_{n'} = \emptyset$  and  $(p, m', \emptyset, \alpha) \in u_{n'}.\Omega$ . As  $u_{n'}$  is a correct node and follows our algorithm, and as  $\alpha \notin Z$ , the only possibility to have  $(p, m', \emptyset, \alpha) \in u_{n'}.\Omega$ , according to Rule 1, is that  $u_{n'} = p$  and  $m' = p.m_0$ , which contradicts our initial hypothesis. Thus, the result. ◀

► **Lemma 4.** *For any two correct nodes  $p$  and  $q$ ,  $\forall m' \neq p.m_0$ , we have  $\text{Count}(q, p, m') \leq |Z|$ .*

**Proof.** Suppose that, at some point, there exist  $q$ ,  $p$  and  $m'$  such that  $\text{Count}(q, p, m') > |Z|$ . It implies that there exists  $\alpha \notin Z$  such that  $(p, m', \alpha) \in q.\text{Acc}_0$ . According to Lemma 3, this is impossible. Thus,  $\text{Count}(q, p, m') \leq |Z|$ . ◀

► **Lemma 5.** *Let  $p$  be a correct node. For any integer  $\alpha_0$ , there exists  $\alpha \geq \alpha_0$  such that  $p$  multicasts a set  $\Omega$  containing  $(p, p.m_0, \emptyset, \alpha)$ .*

**Proof.** According to the algorithm, either we initially have  $p.\alpha \geq \alpha_0$ , or we eventually have  $p.\alpha \geq \alpha_0$ . Thus, according to Rule 1,  $p$  eventually adds  $\{(p, p.m_0, \emptyset, \alpha)\}$  to  $p.\Omega$ , with  $\alpha \geq \alpha_0$ .  
Then, according to Rule 2,  $p$  multicasts a set  $\Omega$  containing  $(p, p.m_0, \emptyset, \alpha)$ . ◀

► **Lemma 6.** *Let  $p$  and  $q$  be two correct nodes. Let  $\alpha$  be an integer. Suppose that  $p$  multicasts a set  $\Omega$  containing  $(p, p.m_0, \emptyset, \alpha)$  at time  $t$ . Suppose that there exists a dynamic path  $(u_1, \dots, u_n)$ , starting after  $t$ , such that  $p = u_1$ ,  $q = u_n$ , and  $\forall i \in \{1, \dots, n\}$ ,  $u_i$  is correct. Then, eventually, we have  $(p, p.m_0, \{u_1, \dots, u_{n-1}\}, \alpha) \in q.\Omega$ .*

**Proof.** Let us prove the following property  $\mathcal{P}_i$  by induction,  $\forall i \in \{1, \dots, n\}$ :  $u_i$  eventually multicasts a set  $\Omega$  containing  $(p, p.m_0, \{u_1, \dots, u_{i-1}\}, \alpha)$ .

- $\mathcal{P}_1$  is true, as  $p$  multicasts a set  $\Omega$  containing  $(p, p.m_0, \emptyset, \alpha)$ .
- Suppose that  $\mathcal{P}_i$  is true, for some  $i \in \{1, \dots, n-1\}$ . Then,  $u_{i+1}$  eventually receives a set  $\Omega$  from  $u_i$  containing  $(p, p.m_0, \{u_1, \dots, u_{i-1}\}, \alpha)$ , and adds  $(p, p.m_0, \{u_1, \dots, u_i\}, \alpha)$  to  $u_i.\Omega$ . Thus, according to Rule 2,  $u_i$  multicasts a set  $\Omega'$  containing  $(p, p.m_0, \{u_1, \dots, u_i\}, \alpha)$ , and  $\mathcal{P}_{i+1}$  is true.

As  $\mathcal{P}_n$  is true,  $u_n = q$  eventually multicasts a set  $\Omega$  containing  $(p, p.m_0, \{u_1, \dots, u_{n-1}\}, \alpha)$ . According to the algorithm, it implies that  $(p, p.m_0, \{u_1, \dots, u_{n-1}\}, \alpha) \in q.\Omega$ . ◀

► **Lemma 7.** *Let  $p$  and  $q$  be two correct nodes. Let  $\alpha$  be an integer. Suppose that  $p$  multicasts a set  $\Omega$  containing  $(p, p.m_0, \emptyset, \alpha)$  at time  $t$ . Then, eventually, we have  $(p, p.m_0, \alpha) \in q.\text{Acc}_0$ .*

**Proof.** Let  $\{S_1, \dots, S_n\}$  be the set of node sets  $S \in \text{Dyn}(p, q, t)$  that contain no Byzantine node. Similarly, let  $\{X_1, \dots, X_{n'}\}$  be the set of node sets  $X \in \text{Dyn}(p, q, t)$  that contain at least one Byzantine node.

Let us suppose that  $MinCut(\{S_1, \dots, S_n\}) \leq k$ . Then, there exists a node set  $C \in Cut(\{S_1, \dots, S_n\})$  such that  $|C| \leq k$ . Let  $C' = C \cup B$  (where  $B$  is the set of Byzantine nodes). Thus,  $C' \in Cut(\{S_1, \dots, S_n\} \cup \{X_1, \dots, X_n\}) = Cut(Dyn(p, q, t))$ , and  $|C'| \leq 2k$ . Thus,  $MinCut(Dyn(p, q, t)) \leq 2k$ , which contradicts our hypothesis. Therefore,  $MinCut(\{S_1, \dots, S_n\}) > k$ .

$\forall S = \{v_1, \dots, v_n\} \in Dyn(p, q, t)$ ,  $(p, v_1, \dots, v_n, q)$  is a dynamic path. Therefore, according to Lemma 6,  $\forall S \in \{S_1, \dots, S_n\}$ , we eventually have  $(p, p.m_0, \{p\} \cup \{v_1, \dots, v_n\}, \alpha) \in q.\Omega$ . Thus, according to the algorithm, we eventually have  $(p, p.m_0, \alpha) \in q.Acc_0$ . ◀

► **Lemma 8.** *Let  $p$  and  $q$  be two correct nodes.  $\forall \alpha_0$ , there exists  $\alpha \geq \alpha_0$  such that we eventually have  $(p, p.m_0, \alpha) \in q.Acc_0$ .*

**Proof.** The result follows from Lemma 5 and Lemma 7. ◀

► **Theorem 9.** *For any two correct nodes  $p$  and  $q$ , there exists a time  $t$  such that, after  $t$ , (1) there exists no  $m' \neq p.m_0$  such that  $(p, m') \in q.Acc$ , and (2)  $(p, p.m_0) \in q.Acc$ .*

**Proof.** According to Lemma 4,  $\forall m' \neq p.m_0$ ,  $Count(q, p, m') \leq |Z|$ . According to Lemma 8 we eventually have  $Count(q, p, p.m_0) \geq |Z| + 1$ . Thus, the result, according to Rule 5 of our algorithm. ◀

## 7 Conclusion

In this paper, we provided the first self-stabilizing and Byzantine-resilient algorithm for reliable communication in dynamic networks, and proved its correctness. To go further, one could consider more probabilistic settings: random positions for Byzantine nodes (each node may have a given probability to be Byzantine), random evolution of the dynamic graph... One could also consider permanent failures at the level of communication channels (e.g., regularly and randomly dropping messages). Finally, an interesting open question would be to consider the time and space complexity of solving this problem, and see if some optimizations could be made w.r.t. these metrics.

---

## References

- 1 The Heartbleed Bug (<http://heartbleed.com>).
- 2 Anish Arora and Sandeep S. Kulkarni. Component based design of multitolerant systems. *IEEE Trans. Software Eng.*, 24(1):63–78, 1998. doi:10.1109/32.663998.
- 3 B. Awerbuch, R. Curtmola, D. Holmer, C. Nita-Rotaru, and H. Rubens. ODSBR: An on-demand secure byzantine resilient routing protocol for wireless ad hoc networks. *ACM Transactions on Information and System Security*, 11:18:1–18:35, 2007.
- 4 Vartika Bhandari and Nitin H. Vaidya. On reliable broadcast in a radio network. In Marcos Kawazoe Aguilera and James Aspnes, editors, *PODC*, pages 138–147. ACM, 2005. doi:10.1145/1073814.1073841.
- 5 Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. Time-varying graphs and dynamic networks. *International Journal of Parallel, Emergent and Distributed Systems*, 27(5):387–408, 2012.
- 6 Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *OSDI*, pages 173–186, 1999. doi:10.1145/296806.296824.
- 7 D. Dolev. The Byzantine generals strike again. *Journal of Algorithms*, 3(1):14–30, 1982.
- 8 Danny Dolev, Cynthia Dwork, Orli Waarts, and Moti Yung. Perfectly secure message transmission. *J. ACM*, 40, January 1993.

- 9 Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000. URL: <http://www.cs.bgu.ac.il/~7Edolev/book/book.html>.
- 10 Vadim Drabkin, Roy Friedman, and Marc Segal. Efficient Byzantine broadcast in wireless ad-hoc networks. In *DSN*, pages 160–169. IEEE Computer Society, 2005. doi:10.1109/DSN.2005.42.
- 11 Chiu-Yuen Koo. Broadcast in radio networks tolerating Byzantine adversarial behavior. In Soma Chaudhuri and Shay Kutten, editors, *PODC*, pages 275–282. ACM, 2004. doi:10.1145/1011767.1011807.
- 12 Sandeep S. Kulkarni and Anish Arora. Compositional design of multitolerant repetitive byzantine agreement. In *Foundations of Software Technology and Theoretical Computer Science, 17th Conference, Kharagpur, India, December 18-20, 1997, Proceedings*, pages 169–183, 1997. doi:10.1007/BFb0058030.
- 13 Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982. doi:10.1145/357172.357176.
- 14 R. Lippmann, K. Ingols, C. Scott, and K. Piwowarski. Validating and restoring defense in depth using attack graphs. *IEEE Military Communications Conference*, 2006.
- 15 Alexandre Maurer and Sébastien Tixeuil. Limiting Byzantine influence in multihop asynchronous networks. In *Proceedings of the 32nd IEEE International Conference on Distributed Computing Systems (ICDCS 2012)*, pages 183–192, June 2012.
- 16 Alexandre Maurer and Sébastien Tixeuil. On Byzantine broadcast in loosely connected networks. In *Proceedings of the 26th International Symposium on Distributed Computing (DISC 2012)*, volume 7611 of *Lecture Notes in Computer Science*, pages 183–192. Springer, 2012.
- 17 Alexandre Maurer and Sébastien Tixeuil. A scalable Byzantine grid. In *Proceedings of the 14th International Conference on Distributed Computing and Networking (ICDCN 2013)*, volume 7730 of *Lecture Notes in Computer Science*, pages 87–101. Springer, 2013.
- 18 Alexandre Maurer and Sébastien Tixeuil. Self-stabilizing byzantine broadcast. In *33rd IEEE International Symposium on Reliable Distributed Systems, SRDS 2014, Nara, Japan, October 6-9, 2014*, pages 152–160. IEEE Computer Society, 2014. doi:10.1109/SRDS.2014.10.
- 19 Alexandre Maurer, Sébastien Tixeuil, and Xavier Défago. Communicating reliably in multihop dynamic networks despite byzantine failures. In *34th IEEE Symposium on Reliable Distributed Systems, SRDS 2015, Montreal, QC, Canada, September 28 - October 1, 2015*, pages 238–245, 2015. doi:10.1109/SRDS.2015.10.
- 20 Henrique Moniz, Nuno Ferreira Neves, and Miguel Correia. Turquoise: Byzantine consensus in wireless ad hoc networks. *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2010)*, 2010.
- 21 Mikhail Nesterenko and Sébastien Tixeuil. Discovering network topology in the presence of Byzantine faults. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 20(12):1777–1789, December 2009. doi:10.1109/TPDS.2009.25.
- 22 Andrzej Pelc and David Peleg. Broadcasting with locally bounded Byzantine faults. *Inf. Process. Lett.*, 93(3):109–115, 2005. doi:10.1016/j.ipl.2004.10.007.



# Fast Deterministic Algorithms for Highly-Dynamic Networks

**Keren Censor-Hillel**

Technion, Haifa, Israel  
ckeren@cs.technion.ac.il

**Neta Dafni**

Technion, Haifa, Israel  
netad@cs.technion.ac.il

**Victor I. Kolobov**

Technion, Haifa, Israel  
tkolobov@cs.technion.ac.il

**Ami Paz**

Faculty of Computer Science, Universität Wien, Austria  
ami.paz@univie.ac.at

**Gregory Schwartzman**

Japan Advanced Institute of Science and Technology, Ishikawa, Japan  
greg@jaist.ac.jp

---

## Abstract

This paper provides an algorithmic framework for obtaining fast distributed algorithms for a highly-dynamic setting, in which *arbitrarily many* edge changes may occur in each round. Our algorithm significantly improves upon prior work in its combination of (1) having an  $O(1)$  amortized time complexity, (2) using only  $O(\log n)$ -bit messages, (3) not posing any restrictions on the dynamic behavior of the environment, (4) being deterministic, (5) having strong guarantees for intermediate solutions, and (6) being applicable for a wide family of tasks.

The tasks for which we deduce such an algorithm are maximal matching,  $(\text{degree} + 1)$ -coloring, 2-approximation for minimum weight vertex cover, and maximal independent set (which is the most subtle case). For some of these tasks, node insertions can also be among the allowed topology changes, and for some of them also abrupt node deletions.

**2012 ACM Subject Classification** Theory of computation

**Keywords and phrases** dynamic distributed algorithms

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2020.28

**Related Version** A full version of the paper is available at <https://arxiv.org/abs/1901.04008>.

**Funding** *Keren Censor-Hillel*: This project has received funding from the European Union's Horizon 2020 Research And Innovation Program under grant agreement no.755839.

*Ami Paz*: We acknowledge the Austrian Science Fund (FWF) and netIDEE SCIENCE project P 33775-N.

*Gregory Schwartzman*: This work was supported by JSPS Kakenhi Grant Number JP19K20216 and JP18H05291.

**Acknowledgements** The authors are indebted to Yannic Maus for invaluable discussions which helped us pinpoint the exact definition of fixing in dynamic networks that we eventually use. We also thank Juho Hirvonen for comments about an earlier draft of this work, and Shay Solomon for useful discussions about his work in [1, 2]. We thank Hagit Attiya and Michal Dory for their input about related work.



© Keren Censor-Hillel, Neta Dafni, Victor I. Kolobov, Ami Paz, and Gregory Schwartzman; licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 28; pp. 28:1–28:16

Leibniz International Proceedings in Informatics



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

## 1 Introduction

We present a family of deterministic distributed algorithms that rapidly fix solutions for fundamental tasks even in a highly-dynamic environment. Specifically, we provide algorithms for maximal matching,  $(\text{degree} + 1)$ -coloring, 2-approximation for the minimum *weighted* vertex cover (2-MWVC), and maximal independent set (MIS). We further show that for some of these tasks, fast fixing is also possible with node insertions and deletions. Here, we consider the severe case of *abrupt* deletions, where a deleted node does not have a chance to inform its neighbors about its upcoming departure from the system.

Our algorithms enjoy the combination of (1) having an  $O(1)$  amortized time complexity, (2) using only  $O(\log n)$ -bit messages, (3) not posing any restrictions on the dynamic behavior of the environment and in particular not requiring topology changes to be spaced in time, (4) being deterministic, (5) having strong guarantees for intermediate solutions, and (6) being applicable for a wide family of tasks. In recent years, there has been much progress on distributed dynamic algorithms, achieving different combinations of the above promises. Our algorithms significantly improve upon all prior work by that they guarantee the combination of all the above properties. We elaborate upon – and compare to – prior work in Section 1.4.

We stress that as opposed to centralized dynamic data structures, not posing any restrictions on the dynamic behavior of the environment is vital in the distributed setting, as the input graph is the communication graph itself. More concretely, in centralized dynamic data structures when multiple topology changes occur, we can simply handle them one by one. However, in our setting, nodes cannot communicate over a deleted edge, and so we cannot sequentially apply an independent update algorithm for each topology change – an edge deletion affects the communication already when it *happens*, not only when it is *handled*.

### 1.1 Motivation

Each of the aforementioned problems is a locally-checkable labeling (LCL) problem. The notion of an LCL is a celebrated concept in distributed computing, first defined by Naor and Stockmeyer [32] in order to capture tasks in which nodes can efficiently detect inconsistencies, motivated by the unstable nature of distributed systems. Since the publication of this pioneering work, the complexity of solving tasks that can be described as LCLs has been extensively studied in the distributed setting. We ask the following question, paraphrased in correspondence with the title of [32]:

*Question: What can be fixed locally?*

We begin by recalling the definition of LCLs of [32], restricting our attention to LCLs with *radius*  $r = 1$ . A *centered star* is a pair  $(H, s)$  where  $H$  is a star graph and  $s$  is its center. An LCL  $\mathcal{L}$  is a tuple  $(\Sigma, \Gamma, \mathcal{C})$ , where  $\Sigma$  is a set of *input labels*,  $\Gamma$  is a set of *output labels*, and  $\mathcal{C}$  is a set of *locally consistent labelings*. Each element of  $\mathcal{C}$  is a centered star, with a label in  $\Sigma \times \Gamma$  for each of its nodes.<sup>1</sup>

A labeling  $\lambda : V \rightarrow \Sigma \times \Gamma$  is called  $\mathcal{L}$ -legal for a graph  $G = (V, E)$ , if for every  $v \in V$ , there exists a centered star  $(H, s)$  in  $\mathcal{C}$  with a label-pair at each node, which is consistent with  $\lambda$  in the following sense: there exists a mapping  $\pi$  that maps the star centered at  $v$  in  $G$  into  $(H, s)$ , with  $\pi(v) = s$ , such that for every node  $w$  in the star centered at  $v$ , the label-pair given by  $\lambda$  is the same as the label-pair of the node  $\pi(w)$  in  $(H, s)$ .

---

<sup>1</sup> In the work of Naor and Stockmeyer [32] the set of labels  $\Sigma$  has a fixed size, while here we omit this limitation in order to give more power to the labelings. However, algorithmically, we always keep the size of messages small even when labels are large, by sending only pieces of them.



As explained in [32], the set  $\mathcal{C}$  defines allowed labels for neighborhoods, as opposed to defining a set of forbidden ones. If the LCL has no inputs, then one can simply choose a default input label, i.e.,  $|\Sigma| = 1$ . An algorithm that solves the problem defined by an LCL  $\mathcal{L}$  is an algorithm whose output on a graph  $G$  is an  $\mathcal{L}$ -legal labeling.

**Not all LCLs are easily fixable.** The following variant of the *sinkless orientation* problem [13] is an example of an LCL problem that is not easily fixable. Each node has a label that corresponds to an orientation of its edges, such that labels at endpoints of an edge are consistent, and such that there is no node of degree greater than 1 that is a sink, i.e., has no outgoing edge. It is easy to verify that every graph has a valid labeling<sup>2</sup>, and that this is an LCL. To see that this LCL cannot be fixed within an amortized complexity of  $O(1)$ , consider a graph on  $n$  nodes that evolves dynamically, creating two paths of roughly  $n/2$  nodes each. Each path must be oriented consistently with a single sink in one of its endpoints. Inserting an edge between the sinks of the two paths forces the orientation of all of the edges in one of the sub-paths to flip, which takes  $\Omega(n)$  rounds. Deleting this edge induces again two paths with a single sink each, and repeating the process of inserting an edge between the new sinks and deleting it causes a linear number of rounds that can be attributed to only two topology changes, which implies an amortized time of  $\Omega(n)$ . This holds even if topology changes do not happen concurrently, and even if the messages can be of arbitrarily large size.

## 1.2 The challenges

For any LCL problem we address, we assume that the system begins with a globally correct labeling, and thus what an algorithm needs to do as a consequence of topology changes is to have the affected nodes update their labels. Naturally, for some problems, the update procedure may also require that a node updates the labels of its neighbors (more precisely, this is accomplished via sending messages to its neighbors requiring them to update their labels). For example, in a solution for maximal matching this might occur when an edge that is in the matching is deleted, and its endpoints need to match themselves to other neighbors. At a first glance, this may sound as a simple and straightforward approach for fixing matchings and problems of local flavor. However, this approach turns out to be far from trivial, and below we describe multiple key challenges that we must overcome in order to implement it successfully.

**(1) Defining *fixing* and *amortized complexity*.** We need to define what *fixing the solution* means. We aim for our algorithm to work in a very harsh setting, in which it might be the case that there are so many topology changes that we never actually obtain a globally correct labeling, but still we maintain strong guarantees for intermediate labelings. Notice that this is in stark contrast to centralized dynamic data structures, which can always consider globally correct solutions since topology changes may be handled one-at-a-time because they only affect the input and not the computation itself. This is also the case for the majority of previous distributed algorithms: they are designed under the assumption that topology changes are spaced well enough in time so that it is possible to obtain a globally correct solution before the next topology change happens.

---

<sup>2</sup> If  $G$  is a tree, choose an arbitrary root and orient the edges away from the root. Otherwise, choose a cycle in  $G$  and orient its edges cyclically, then imagine contracting its nodes into a single super-node and orient edges towards this super-node along some spanning tree, and orient other edges arbitrarily.

**(2) Coping with concurrent fixing with a timestamp mechanism.** Because we might need a node to change the labels of its neighbors and not only its own label in order to fix the solution, we make sure that concurrent fixing always happens for nodes that are not too close, and other nodes wait even if their labeled stars are not yet correct (e.g., to avoid two nodes  $u, v$ , trying to get matched to the same node  $w$  concurrently). To this end, our method is to assign a timestamp to each node involved in a change, and *fix* a node only if its timestamp is a local minimum in some short-radius neighborhood, thus avoiding conflicting concurrent fixes. We call such a node *active*.

**(3) Detecting and aborting conflicting timestamps.** Such a timestamp mechanism alone is still insufficient: the uncontrolled number of topology changes may, for example, suddenly connect two nodes that were previously far enough so that they could become active simultaneously, but after concluding that they can both become active, an edge insertion now makes them part of the same short-radius neighborhood. We carefully take care of such cases where our timestamps have been cheated by the topology changes, by detecting such occurrences and *aborting the fixing*, without harming the amortized complexity guarantees.

**(4) Bounding the size of timestamps to cope with message size restrictions.** Finally, the restriction on the size of messages forbids unbounded timestamps, despite an unbounded number of rounds (e.g., times). To resolve this issue, we utilize ideas from the literature on shared memory algorithms, e.g., [3], for deterministically hashing the timestamps into a small bounded domain so that the nodes can afford sending a hashed timestamp in a single small message, and we do so in a way that preserves the total order over timestamps.

### 1.3 Our contributions

Our main contribution is thus deterministic dynamic distributed fixing algorithms for several fundamental problems. Our algorithms share a common approach, and only minor modifications that are specific to each labeling are required. In some cases we can also handle a node insertion/deletion, which is a-priori possibly harder to deal with, because it may affect more nodes while in the amortized analysis we count it as a single topology change.

The following theorem summarizes the end-results, which hold in a model with an unbounded number of topology changes that may occur concurrently, and when only a logarithmic number of bits can be sent in a message.

► **Theorem 1.** *There is a deterministic dynamic distributed fixing algorithm for  $(\text{degree}+1)$ -coloring and for a 2-approximation of a minimum weight vertex cover, which handles edge insertions/deletions and node insertions in  $O(1)$  amortized rounds.*

*There are deterministic dynamic distributed fixing algorithms for maximal matching,  $(\Delta+1)$ -coloring (where  $\Delta$  is the maximum node degree) and MIS, which handle edge/node insertions/deletions in  $O(1)$  amortized rounds.*

Section 3 shows our algorithm for maximal matching. This is developed and modified in the full version of the paper to present our 2-MWVC algorithm. We mention that the labeling for the solution of 2-MWVC that we maintain is not the naïve one that only indicates which nodes are in the cover, but rather contains information about dual variables that correspond to edge weights, and allow the fast fixing.

Section 4 gives our algorithm for MIS. In the MIS case, the restriction of message size imposes an additional, huge difficulty. The reason is that if an MIS node  $v$  needs to leave the MIS because an edge is inserted between  $v$  and some other MIS node  $u$ , then all other

neighbors of  $v$  who were previously not in the MIS are now possibly not covered by an MIS neighbor. Yet, they cannot all be moved into the MIS, as they may have an arbitrary topology among them. With unbounded messages this can be handled using very large neighborhood information but such an approach is ruled out by the restriction of  $O(\log n)$ -bit messages.

Nevertheless, we prove that with some modifications to our algorithmic approach, we can also handle MIS without the need to inform nodes about entire neighborhoods. The road we take here is that instead of fixing its neighborhood, a node tells its neighbors that they should become active themselves in order to fix their labeled stars. On the surface, this would entail an unacceptable overhead for the amortized complexity that is proportional to the degree of the node. The crux in our algorithm and analysis is in blaming previous topology changes for such a situation – for every node  $u$  in the neighborhood of  $v$  which is only dominated by  $v$ , there is a previous topology change (namely, an insertion of an edge  $\{u, w\}$ , where  $w$  may or may not be  $v$ ) for which we did not need to fix the label of  $w$ . This accounting argument allows us to amortize the round complexity all the way down to  $O(1)$ , and the same technique is utilized to handle node insertions and deletions. In the full version of the paper, we present our algorithm for  $(\text{degree} + 1)$ -coloring, as well as a generalization of our algorithm, by defining a family of graph labelings, in the flavor of the LCL definition, which can all be fixed in constant amortized time.

## 1.4 Related work

The end results of our work provide fast fixing for fundamental graph problems, whose static algorithmic complexity has been extensively studied in the distributed setting. A full overview of the known results merits an entire survey paper on its own (see, e.g., [8, 35]). An additional line of beautiful work studies the landscape of distributed complexities of LCL problems, and the fundamental question of using randomness (see, e.g., [5, 6, 14, 17, 18, 23]).

For dynamic distributed computing, there is a rich history of research on the important paradigm of *self-stabilization* (see, e.g., the book [20]) and in particular on symmetry breaking (see, e.g., the survey [24]). Related notions of error confinement and fault-local mending have been studied in [4, 30, 31]. Our model greatly differs from the above. There are many additional models of dynamic distributed computation (e.g., [12, 29]), which are very different from the one we consider in this paper.

Some of the oldest works in similar models to ours are [22, 26], who provide algorithms for distance-related tasks. Constant-time algorithms were given in [28] for symmetry-breaking problems assuming unlimited bandwidth and a single topology change at a time. The work of [15], provides a randomized algorithm that uses small messages to fix an MIS in  $O(1)$ -amortized update time for a non-adaptive oblivious adversary, still assuming a single change at a time. The latter left as an open question the complexity of fixing an MIS in the sequential dynamic setting. This was picked up in [1, 2, 21, 25], giving the first non-trivial sequential MIS algorithms, which were recently revised and improved [10, 19]. Specifically, the algorithm of [1] achieves an  $O(\min\{\Delta, m^{3/4}\})$  amortized message complexity and  $O(1)$ -amortized round complexity and adjustment complexity (the number of vertices that change their output after each update) for an adaptive non-oblivious adversary in the distributed setting. However, they handle only a single change at a time, and sometimes need to know the number of edges, which is global knowledge that our work avoids assuming. In fact, if one is happy with restricting the algorithm to work only in a model with a single topology change at a time, then sending timestamps is not required, so  $O(1)$ -bit messages suffice in our algorithm for MIS, resembling what [1] obtains.

[33] provides a neat log-starization technique, which translates logarithmic static distributed algorithms into a dynamic setting such that their amortized time complexity becomes  $O(\log^* n)$ . This assumes a single change at a time and large messages. [34] shows that maximal matching have  $O(1)$  amortized complexity, even when counting messages and not only rounds, but assuming a single change at a time.

The  $(\Delta + 1)$ -coloring algorithm of [9] also implies fixing in a self-stabilizing manner – after the topology stops changing, only  $O(\Delta + \log^* n)$  rounds are required in order to obtain a valid coloring, where  $\Delta$  bounds the degrees of all the nodes at all times.

Perhaps the setting most relevant to ours is the one studied in [7], who also address a very similar highly-dynamic setting. They insightfully provide fast dynamic algorithms for a wide family of tasks, which can be decomposed into packing and covering problems, in the sense that a packing condition remains true when deleting edges and a covering condition remains true when inserting edges. For example, MIS is such a problem, with independence and domination being the packing and covering conditions, respectively. An innovative contribution of their algorithms is providing guarantees also for intermediate states of the algorithm, that is, guarantees that hold even while the system is in the fixing process. They show that the packing property holds for the set of edges that are present throughout the last  $T$  rounds, and that the covering property holds for the set of edges that are present in either of the last  $T$  rounds, for  $T = O(\log n)$ . Moreover, their algorithms have correct solutions if a constant neighborhood of a node does not change for a logarithmic number of rounds. Our algorithm guarantees correctness of labeled stars for nodes for which any topology change touching their neighborhood has already been handled. In comparison with their worst-case guarantee of  $O(\log n)$  rounds for a correct solution, our algorithm only gives  $O(n)$  rounds in the worst case. However, our amortized complexity is  $O(1)$ , our messages are of logarithmic size, and our algorithm is deterministic, while the above is randomized with messages that can be of polylogarithmic size. In addition, a recent work [16] studies subgraph problems in the same model described in our paper.

A different definition of local fixability [11, Appendix A], suitable for *sequential* dynamic data structures, requires a node to be able to fix the solution by changing only its own state. While this captures tasks such as coloring, and is helpful in the sequential setting for avoiding the need to update the state of all neighbors of a node, in the distributed setting we can settle for a less restrictive definition, as a single communication round suffices for updating states of neighbors, if needed. Our algorithmic framework captures a larger set of tasks: notably, we provide an algorithm for MIS, while [11] prove that it does not fall into their definition. In addition, [11, Section 7] raises the question of fixing (in the sequential setting) problems that are in P-SLOCAL<sup>3</sup> [23]. Notably, this class contains approximation tasks, and indeed for some approximation ratios we can apply our framework: Our algorithm has the flavor of sequentially iterating over nodes and fixing the labels in their neighborhood, with the additional power of the distributed setting that allows it to work concurrently on nodes that are not too close. This also resembles the definition of orderless local algorithms [27], although a formal definition for the case of fixing does not seem to be simpler than ours.

---

<sup>3</sup> Roughly speaking, SLOCAL( $t$ ) is the class of problems that admit solutions by an algorithm that iterates over all the nodes of the graph, and assigns a solution to each node based on the structure of its  $t$ -neighborhood and solutions already assigned to nodes in this neighborhood. P-SLOCAL is the class SLOCAL(polylog  $n$ ).

## 2 Model

We assume a synchronous network that starts as an empty graph on  $n$  nodes and evolves into the graph  $G_i = (V_i, E_i)$  at the beginning of round  $i$ ; in most of our algorithms, one can alternatively assume any graph as the initial graph, as long as the nodes start with a labeling that is globally consistent for the problem in hand. In some cases, we also allow node insertion or deletion, and then  $n$  serves as a universal upper bound on the number of nodes in the system. Each node is aware of its unique id, the edges it is a part of, its weight if there is one, and of  $n$ . In addition, the nodes have a common notion of time, so the execution is synchronous. New nodes do not know the global round number. (We mention that in our algorithms it is sufficient for each node to know the round number modulo  $15n$ , and a new node can easily obtain this value from its neighbors, so we implicitly assume all nodes have this knowledge.)

In each round, each node receives *indications* about the topology changes that occurred to its incident edges. We stress that the indications are a posteriori, i.e., the nodes get them only after the changes occur, and thus cannot prepare to them in advance (these are called *abrupt* changes). After receiving the indications and performing local computation, each node can send messages of  $O(\log n)$  bits to each of its neighbors.

We work in a distributed setting where each node stores its own label. A distributed fixing algorithm should update the labels of the nodes in a way that corrects the labeled stars that become incorrect due to topology changes. Naturally, for a highly-dynamic setting, we do not require a global consistent labeling in scenarios in which the system is undergoing many topology changes.

We consider four classical graph problems. In the *maximal matching* problem, the nodes have to mark a set of edges such that no two intersect, and such that no edge can be added to the set without violating this condition. In *minimum weight vertex cover* (MWVC), the nodes start with weights, and the goal is to choose a set of nodes that intersect all the edges, and have the minimum weight among all such sets; we will be interested in the 2-approximation variant of the problem, where the nodes choose a set of weight at most twice the minimum. Finally, in the *maximal independent set* (MIS) problem, the nodes must mark a set of nodes such that no two adjacent nodes are chosen, and such that no node can be added to the set.

**The complexity of distributed fixing algorithms.** When the labels of a star become inconsistent due to changes, a distributed fixing algorithm will perform a fixing process, which ends when the labels are consistent again, or when other changes occur in this star. The *worst-case round complexity* of a distributed fixing algorithm is the maximum number of rounds such a fixing process may take.

In our algorithms, it could be that it takes a while to fix some star, but we can argue that this is because other stars are being fixed. We measure this progress with a definition of the amortized round complexity.

When studying *centralized* algorithms for dynamic graphs, the amortized complexity measure is typically defined by an *aggregate analysis*, i.e., considering the time when the fixing process ends, and dividing the number of computation steps taken so far by the number of changes that occurred. The natural generalization of this definition to the distributed setting could be to take a time when the graph labeling is globally correct, and divide the number of rounds occurred so far by the number of changes the network had undergone. The first and most eminent problem in such a definition is that it requires a time when the *global solution* is correct, which is something that we cannot demand in a highly-dynamic

environment. The second problem with it is that the adversary can fool this complexity measure, by doing nothing for some arbitrary number of rounds in which the graph is correct, while the algorithm still gets charged for these rounds.

To overcome the above problems, we define the amortized round complexity as follows. Starting from round 0, in which the labeling is consistent for all stars, we consider the situation in each round  $i$ . We denote by  $\text{incorrect}(i)$  the number of rounds until round  $i$  in which there exists at least one inconsistent star. These are the computation rounds for which we charge the algorithm. Notice that we do not count only communication rounds in order to prevent an algorithm that cheats by doing nothing.<sup>4</sup> We denote by  $\text{changes}(i)$  the number of changes which occurred until round  $i$ . We say that an algorithm has an *amortized round complexity*  $k$  if for every  $i$  with  $\text{changes}(i) > 0$ , we have  $\text{incorrect}(i)/\text{changes}(i) \leq k$ . This definition captures the *rate* at which changes are handled, in a way that generalizes the sequential definition.

**Guarantees of our algorithm.** Our algorithms have an  $O(1)$  amortized fixing time, and in addition, they have additional desired progress properties. First, our algorithms guarantee a worst-case complexity of  $O(n)$ , which implies that repeated changes far from a given star will not postpone it from being fixed for too long. Moreover, if a labeled star is consistent and no topology change touches its neighborhood, then it remains consistent. Thus, our algorithm has strong guarantees also for intermediate solutions.

### 3 An $O(1)$ amortized dynamic algorithm for maximal matching

The solution to the maximal matching problem at any given time is determined according to the labels of the nodes. A label of a node  $v$  can be either `unmatched` or `matched-to- $u$` , indicating that  $v$  is unmatched, or is matched to  $u$ , respectively. Each node starts with the label `unmatched`. Alternatively, one can assume any graph as the initial graph, as long as the nodes start with a legal maximal matching solution. We prove the following.

► **Theorem 2.** *There is a deterministic dynamic distributed fixing algorithm for maximal matching which handles edge insertions/deletions in  $O(1)$  amortized rounds.*

**Proof.** First, we assume that all nodes start with an initial globally consistent solution.

**The setup:** We denote  $\gamma = 5$ .

Let  $F_i$  be a set of edge changes (insertions/deletions) that occur in round  $i \geq 0$  (for convenience, the first round is round 0). With each change in  $F_i$ , we associate two *timestamps* such that a total order is induced over the timestamps as follows: for an edge  $e = \{u, v\}$  in  $F_i$ , we associate the timestamp  $ts = (i, u, v)$  with node  $u$ , and the timestamp  $(i, v, u)$  with node  $v$ . Since  $u$  and  $v$  start round  $i$  with an indication of  $e$  being in  $F_i$ , both can deduce their timestamps at the beginning of round  $i$ . We say that a node  $v$  is the *owner* of the timestamps that are associated with it. In each round, a node only stores the largest timestamp that it owns, and omits the rest.

Notice that timestamps are of unbounded size, which renders them impossible to fit in a single message. To overcome this issue we borrow a technique of [3], and we invoke a deterministic hash function  $H$  over the timestamps, which reduces their size to  $O(\log n)$

<sup>4</sup> One could count also rounds in which the labeling is globally correct if the algorithm chooses to communicate in these rounds. Our algorithm never communicates in such rounds, so such a definition would not change our amortized complexity.



bits, while retaining the total order over timestamps. The reason we can do this is that not every two timestamps can exist in the system concurrently. To this end, we define  $h(i) = i \bmod 3\gamma n$  and  $H(ts) = (h(i), u, v)$  for a timestamp  $ts = (i, u, v)$ , and we define an order  $\prec_H$  over hashed timestamps as the lexicographic order of the 3-tuple, induced by the following order  $\prec_h$  over values of  $h$ . We say that  $h(i) \prec_h h(i')$  if and only if one of the following holds:

- $0 \leq h(i) < h(i') \leq 2\gamma n$ , or
- $\gamma n \leq h(i) < h(i') < 3\gamma n$ , or
- $2\gamma n \leq h(i) < 3\gamma n$  and  $0 \leq h(i') < \gamma n$ .

If two timestamps  $ts = (i, v, u)$ ,  $ts' = (i', v', u')$  are stored in two nodes  $v, v'$  at two times  $i, i'$ , respectively, it holds that  $ts < ts'$  (by the standard lexicographic order) if and only if  $H(ts) \prec_H H(ts')$ . The reason that this holds despite the wrap-around of hashed timestamps in the third bullet above, is the following property that we will later prove: for every two such timestamps, it holds that  $i' - i \leq \gamma n$ . This implies  $h(i) \prec_h h(i')$  whenever  $i < i'$  despite the bounded range of the function  $h$ .

**The algorithm:** In the algorithm, time is chopped up into *epochs*, each consisting of  $\gamma$  consecutive rounds, in a non-overlapping manner. That is, epoch  $j$  consists of rounds  $i = \gamma j, \dots, \gamma(j+1) - 1$ . For every epoch  $j \geq 0$ , we consider a set  $D_j \subseteq V$  of *dirty* nodes at the beginning of each epoch, where initially no node is dirty ( $D_0 = \emptyset$ ). Some nodes in  $D_j$  may become *clean* by the end of the epoch, so at the end of the epoch the set of dirty nodes is denoted by  $D'_j$ , and it holds that  $D'_j \subseteq D_j$ . At the beginning of epoch  $j+1$ , all nodes that receive any indication of an edge in  $F_i$  in the previous epoch are added to the set of dirty nodes, i.e.,  $D_{j+1} = D'_j \cup I_j$ , where  $I_j$  is the set of nodes that start round  $i$  with any indication about  $F_i$ , for any  $\gamma j \leq i \leq \gamma(j+1) - 1$ .

Intuitively, the algorithm changes the labels so that *the labels at the end of the epoch are consistent with respect to the topology that was at the beginning of the epoch*, unless they are labels of dirty nodes or of neighbors of dirty nodes.

The algorithm works as follows. In epoch  $j = 0$ , the nodes do not send any messages, but some of them enter  $I_0$  (if they receive indications of edges in  $F_i$ , for  $0 \leq i \leq \gamma - 1$ ).

Denote by  $N_v^i$  the neighborhood of  $v$  in round  $i$ , denote by  $L_v^i$  the label of  $v$  at the beginning of round  $i$ , before the communication takes place, and denote by  $\hat{L}_v^i$  the label at the end of the round. Unless stated otherwise, the node  $v$  sets  $\hat{L}_v^i \leftarrow L_v^i$  and  $L_v^{i+1} \leftarrow \hat{L}_v^i$ . Now, consider an epoch  $j > 0$ . On round  $\gamma j$  every node  $v \in D_j$  may locally change its label to indicate that it is unmatched, in case the edge between  $v$  and its previously matched neighbor is deleted:

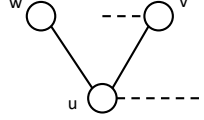
$$L_v^{\gamma j} = \begin{cases} \text{matched-to-}u, & \text{if } \hat{L}_v^{\gamma j-1} = \text{matched-to-}u \text{ and } u \in N_v^{\gamma j} \\ \text{unmatched}, & \text{otherwise} \end{cases} \quad (1)$$

where  $\hat{L}_v^{\gamma j-1}$  is the label that  $v$  has at the *end* of round  $\gamma j - 1 = \gamma(j-1) + 4$ , which, as we describe below, may be different from its label  $L_v^{\gamma j-1}$  at the beginning of the round.<sup>5</sup> Then, the node  $v$  sends  $L_v^{\gamma j}$  to its neighbors. These are the labels for the graph  $G_{\gamma j}$  which the fixing addresses. We stress that the new labels  $L_v^{\gamma j}$  might not form consistent stars. Instead, the nodes update  $L_v^{\gamma j}$  and send it to all neighbors in order to maintain a common graph, with respect to which we show local consistency. As an example, consider a triangle  $w, v, u$ ,

<sup>5</sup> We stress that one can describe our algorithm with labels that can only change at the beginning of a round, but we find the exposition clearer this way.



undergoing the deletion of the edge  $\{w, v\}$  and of another edge connecting  $u$  with some other node (see Figure 1). Suppose that  $w$  immediately tries to fix the labels in its star, according to the fact that the edge  $\{w, v\}$  does not exist, while  $u$  is selected to fix its own star before  $v$ , without knowing of the deletion of the edge  $\{w, v\}$ . Both nodes then simultaneously try to change the label of  $u$ , and it could not be clear what  $u$  should do, and which neighborhood of  $u$  will be corrected.



■ **Figure 1** Dashed lines represent edges that were deleted.

We continue describing the algorithm. On rounds  $\gamma j + 1$  to  $\gamma j + 3$  the nodes propagate the hashed timestamps owned by dirty nodes. That is, on round  $\gamma j + 1$ , each node in  $D_j$  broadcasts its hashed timestamp, and on the following two rounds all nodes broadcast the smallest hashed timestamp that they see (with respect to the order  $\prec_H$ ). Every node  $v$  in  $D_j$  which does not receive a hashed timestamp that is smaller than its own becomes *active*.

On the last round of the epoch,  $\gamma j + 4$ , every active node  $v$  computes the following candidate for a new label, denoting by  $N_v^{\gamma j} = \{u_1, \dots, u_d\}$  the neighborhood it had at round  $\gamma j$ .

$$\ell_v = \begin{cases} \text{matched-to-}u_i, & \text{if } L_v^{\gamma j} = \text{matched-to-}u_i \\ \text{unmatched}, & \text{if } L_v^{\gamma j} = \text{unmatched} \text{ and for every } 1 \leq i \leq d, L_{u_i}^{\gamma j} \neq \text{unmatched} \\ \text{matched-to-}u_i, & \text{if } L_v^{\gamma j} = \text{unmatched} \text{ and } 1 \leq i \leq d \text{ is the smallest index} \\ & \text{for which } L_{u_i}^{\gamma j} = \text{unmatched} \end{cases} \quad (2)$$

Notice that  $v$  has the required information to compute the above, even if additional topology changes occur during the rounds in which timestamps are propagated. Yet, we need to cope with the fact that topology changes may occur also throughout the current epoch and, for example, make active nodes suddenly become too close. For this, we denote by  $T_j \subseteq I_j$  the set of *tainted* nodes who received an indication of a topological change for at least one of their edges during the epoch  $j$ .

Now, only an active node  $v$  which is not in  $T_j$  sets  $\hat{L}_v^{\gamma j+4} \leftarrow \ell_v$  and sends this new label to each neighbor  $u$ . Otherwise, an active node  $v$  that is tainted (i.e., is in  $T_j$ ) aborts and remains dirty for the next epoch. Of course, if nodes  $u$  and  $v$  are neighbors at the beginning of an epoch but not when  $v$  sends the computed label, then  $u$  does not receive this information.

Finally, every active node  $v \notin T_j$ , if  $\hat{L}_v^{\gamma j+4} = \text{matched-to-}u$  then  $u$  updates  $\hat{L}_u^{\gamma j+4} = \text{matched-to-}v$  (note that such  $u$  has the required information since it receives  $\ell_v$ , as otherwise, if by the time that  $\ell_v$  is computed it holds that  $u$  and  $v$  are no longer neighbors, then  $v$  must be tainted). At the end of round  $\gamma j + 4 = \gamma(j + 1) - 1$ , node  $v$  becomes *inactive* and, unless it aborts, is not included in  $D'_j$ , i.e., we initialize  $D'_j = D_j \setminus \{v \mid v \notin T_j \text{ is active in epoch } j\}$  at the end of epoch  $j$ .

**Correctness.** For correctness we claim the following invariant holds at the end of round  $i = \gamma j + 4 = \gamma(j + 1) - 1$ : For every two nodes  $u, v$  that are clean at the end of the epoch and for which  $\{u, v\}$  is an edge in  $G_{\gamma j}$ , it holds that (1) at least one of  $\hat{L}_u^{\gamma j+4}$  and  $\hat{L}_v^{\gamma j+4}$  is not *unmatched* and (2) if  $\hat{L}_u^{\gamma j+4} = \text{matched-to-}v$  then  $\hat{L}_v^{\gamma j+4} = \text{matched-to-}u$ .

We prove the above by induction on the epochs. The base case holds trivially as during the first epoch the labels do not change, and we assume that the nodes start with a legal maximal matching for the initial graph. Now, assume the above invariants hold for epoch  $j - 1$ .

For every two nodes  $u, v$  that are clean at the end of the epoch and for which  $\{u, v\}$  is an edge in  $G_{\gamma j}$ , if their labels do not change during the epoch, then the invariant follows from the induction hypothesis.

If only one of their labels changes, say that of  $v$ , then either  $v$  is active and not tainted or there is a (single) neighbor  $w$  of  $v$  which is active and not tainted and makes  $v$  change its label. In the former case, since the label  $\ell_v$  of  $v$  changes compared to  $L_v^{\gamma j}$ , it does not remain `unmatched` and does not remain `matched-to- $x$`  for some node  $x$ . So the new label  $\ell_v$  must be `matched-to- $y$` , for some node  $y$ . Since the label of  $u$  does not change, we have that  $u \neq y$ , and so if the label of  $u$  is not `unmatched` then it cannot be `matched-to- $v$`  (as otherwise  $L_v^{\gamma j}$  would be `matched-to- $u$`  and so  $\ell_v$  would also be `matched-to- $u$` , thus did not change). In the latter case, if  $v$  changes its label because of the new label  $\ell_w$  that is sent to it by a neighbor  $w$ , then  $\ell_w = \text{matched-to-}v$  and hence the new label of  $v$  is set to `matched-to- $w$` .

Finally, if both of their labels change, then without loss of generality  $v$  is active and not tainted and computes  $\ell_v = \text{matched-to-}u$ , making  $u$  update its label to `matched-to- $v$` . The crucial thing to notice here is that it cannot be the case that a node  $w_v$  changes the label of  $v$  and a different node  $w_u$  changes the label of  $u$  at the same time, because this implies that the distance between  $w_v$  and  $w_u$  is at most 3, in which case either at least one of them aborts due to an edge insertion, or the edge  $\{u, v\}$  is inserted (maybe immediately after being deleted), but then  $v$  and  $u$  are not clean.

Since the invariant holds, we conclude that whenever  $D_j = \emptyset$ , it holds that the labeling is that of a maximal matching for  $G_{\gamma j}$ . Further, what the invariant implies is that some correctness condition holds even for intermediate rounds: at the end of every epoch  $j$ , the entire subgraph induced by the set of nodes that are clean and have all of their neighborhood clean consists of nodes with locally consistent labels.

**Round complexity.** We now prove that the algorithm has an amortized round complexity of  $O(1)$ , by proving  $\text{incorrect}(i) \leq 2\gamma \cdot \text{changes}(i)$  for all  $i$ . First, note that the algorithm communicates in each round where the graph is incorrect, and these communication rounds can be split into epochs, implying  $\text{incorrect}(i) \leq \gamma \cdot \text{epochs}(i)$ , where  $\text{epochs}(i)$  denotes the number of epochs of computation done by the algorithm until round  $i$  (if round  $i$  is the middle of an epoch then it does not affect the asymptotic behavior, so we can safely ignore this partial epoch). On the other hand, the node with minimal timestamp at the beginning of the  $j$ -th epoch becomes active during the epoch, and its timestamp is handled – even if it becomes tainted by a change, the old timestamp is replaced by the new one. So, in each epoch at least one timestamp disappears from the system. Now, since each topology change creates at most two timestamps, we have that the number of timestamps created until round  $i$  is at most  $2 \cdot \text{changes}(i)$ , implying  $\text{epochs}(i) \leq 2 \cdot \text{changes}(i)$ , and the claim follows.

Finally, we show that the timestamps can be represented by  $O(\log n)$  bits. First, we claim that for every two timestamps  $ts = (i, v, u)$  and  $ts' = (i', v', u')$  such that  $ts < ts'$ , that are simultaneously owned by nodes at a given time, it holds that  $i' - i \leq \gamma n$ . Assume otherwise, and consider the first time when this condition is violated by a timestamp  $ts'$ , with respect to a previous timestamp  $ts < ts'$ . This means that the owner  $v$  of  $ts$  does not become active for more than  $n$  epochs. Since up to this point in time there were no violations, in each epoch at least one timestamp was handled, and this was done in the desired order, i.e., all these labels were smaller than  $ts$ . So,  $v$  not becoming active for more than  $n$  epochs can only

happen if at round  $i$  there were more than  $n$  timestamps which were then not yet handled, stored in various nodes. But there are at most  $n$  nodes and each one stores at most one timestamp so the above is impossible. Since  $i' - i \leq \gamma n$ , we have that  $H(ts) \prec_H H(ts')$ , because  $h(i) \prec_h h(i')$ , as argued earlier.

Using the above we can also see that the worst case running time of our algorithm is  $O(n)$ . To see this, fix some node  $v$  with an inconsistent star which does not experience topology changes touching its 1-hop neighborhood for  $(\gamma + 1)n$  rounds. This guarantees that its timestamp does not change throughout these rounds, and after  $\gamma n$  rounds its timestamp must become a local minima. In the following epoch, if no changes occur within its 1-hop neighborhood then its star becomes consistent, which matches the definition of having a worst-case complexity of  $O(n)$ . Further, once a node  $v$  successfully invokes a fixing of its star, the star remains consistently labeled as long as no topology changes touch the 1-hop neighborhood of  $v$ , thus we obtain strong guarantees for intermediate solutions. ◀

For node insertions and deletions, a direct application of the algorithm of Theorem 2 increases the amortized complexity if all neighbors of a changed node (inserted or deleted) become dirty and  $O(\Delta)$  timestamps are associated with this topology change. However, notice that when an edge is inserted, it suffices that *only one* of its endpoints becomes dirty in the algorithm and gets matched to the other endpoint if needed. Hence, if a node is inserted, it suffices that the inserted node becomes dirty, and we do not need all of its neighbors to become so. An only slightly more subtle rule for deciding which nodes become dirty upon a node deletion gives the following.

► **Theorem 3.** *There is a deterministic dynamic distributed fixing algorithm for maximal matching which handles edge/node insertions/deletions in  $O(1)$  amortized rounds.*

**Proof.** We modify the algorithm of Theorem 2 as follows. Upon an insertion of a node  $v$ , the node  $v$  becomes dirty. Upon a deletion of a node  $v$  with neighbors  $\{u_1, \dots, u_d\}$ , only the node  $u_i$ , for  $1 \leq i \leq d$ , that is matched to  $v$  (if there exists such a node) becomes dirty.

The  $O(1)$  amortized round complexity remains, as every topology change induces at most two new timestamps. Correctness still holds because it is not affected by a node insertion, which can be viewed as multiple edge insertions (in terms of correctness, but without paying this cost for the amortized time complexity), and it is not affected by a node deletion because for any other node  $u_j \in \{u_1, \dots, u_d\}$  such that  $j \neq i$  it holds that the deletion of  $v$  does not influence its local consistency. ◀

#### 4 An $O(1)$ amortized dynamic algorithm for MIS

One can use a similar approach in order to obtain an MIS algorithm. However, when a node  $v$  needs to be removed from the MIS due to an edge insertion, a neighbor  $u$  of  $v$  may need to join the MIS if none its other neighbors are in the MIS. One way to do this is to mark all of the neighbors of  $v$  as dirty, but this violates the amortized time complexity as a single topology change may incur too many dirty nodes. Another option is to have  $v$ 's label include neighborhood information such that upon receiving this label, its neighbors know which of them should be moved into the MIS. This would give a simple  $O(1)$  amortized rounds algorithm for MIS, but only if messages are allowed to be large. Instead, we present a labeling that does not contain all the neighborhood information and uses only *small* labels.

To handle the new subset of neighbors that needs to be added to the MIS in the aforementioned example, our approach is to have the active node simply indicate to all of its neighbors that they cannot remain clean and must check for themselves whether they need to change their labels. Of course, such a single topology change may now incur a number of dirty nodes that is the degree of this endpoint, which may be linear in  $n$ .

Yet, we make a crucial observation here: any node that becomes dirty in this manner, can be blamed on a previous topology change in which only one node becomes dirty. This implies a budget, to which we add 2 units for every topology change, and charge either 0, 1, 2, or  $d$  (current node degree) units for each invocation of the fixing function, in a manner that preserves the budget non-negative at all times. Note that due to the accounting argument, here we must start with an empty graph for the amortization to work, unlike previous problems, where we could start with any graph as long as the nodes have labels that indicate a valid solution. Roughly speaking, we rely on the fact that since all nodes that are in the graph start as MIS nodes because there are no edges, then a node switches from being an MIS node to being a non-MIS node only upon an insertion of an edge between two MIS nodes, and the other endpoint of the inserted edge safely remains in the MIS. Note that we impose the rule that an inserted node never makes a node switch from being an MIS node to being a non-MIS node, since the inserted node chooses to become an MIS node only if all of its neighbors are already non-MIS nodes.

► **Theorem 4.** *There is a deterministic dynamic distributed fixing algorithm for MIS, which handles edge/node insertions/deletions in  $O(1)$  amortized rounds.*

**Proof.** We consider labels which are in  $\{\mathbf{true}, \mathbf{false}\}$  and maintain that the set of nodes with the label  $\mathbf{true}$  form an MIS. We start with an empty graph and all labels are  $\mathbf{true}$ . We first define the assignments of  $L_v^i$  and  $\ell_v$  as in assignments (1) and (2) in the algorithm for maximal matching in the proof of Theorem 2. Then, we explain how we modify the algorithm further in order to avoid large messages with neighborhood information.

First, the label for  $L_v^{\gamma_j}$  does not change from the previous round, i.e., assignment (1) is  $L_v^{\gamma_j} = \hat{L}_v^{\gamma_{j-1}+4}$ . For assignment (2) we set  $\ell_v$  to be  $\mathbf{false}$  if  $L_{u_i}^{\gamma_j} = \mathbf{true}$  for some  $u_i \in N_v^{\gamma_j}$  and otherwise we set  $\ell_v$  to be  $\mathbf{true}$ . If  $v$  is active and not in  $T_j$  then it sets  $\hat{L}_v^{\gamma_{j+4}}$  to be  $\ell_v$  and sends this label to all of its neighbors. Notice that this is insufficient for arguing that the labels at the end of the epoch form an MIS if all nodes are clean, for the same reason as in the tricky example above: if  $v$  leaves the MIS due to an edge insertion, its neighbors do not have enough information to decide which of them joins the MIS. To overcome this challenge, we consider an algorithm similar to the one of Theorem 2, with the following modifications.

- (1) When an edge  $e = \{v, u\}$  is deleted, if the labels of both  $u$  and  $v$  are  $\mathbf{false}$  then neither of them becomes dirty, and if only one of them is  $\mathbf{false}$  then only this node becomes dirty.
- (2) When an edge  $e = \{v, u\}$  is inserted, if at least one of the labels of  $u$  and  $v$  is  $\mathbf{false}$  then neither of them becomes dirty, and if both are  $\mathbf{true}$  then only the node with smaller  $ID$  becomes dirty.
- (3) When a node  $v$  is inserted then only  $v$  becomes dirty.
- (4) When a node  $v$  is deleted then a neighbor  $z$  becomes dirty only if its label is  $\mathbf{false}$  and it has no neighbor with a label  $\mathbf{true}$ .

In order for a node  $v$  to indicate that new labels may be needed for its neighbors, we add the following item:

- (5) When an active node  $v$  changes its label to  $\mathbf{false}$ , all of its neighbors marked  $\mathbf{false}$  that do not have a neighbor marked  $\mathbf{true}$  become dirty.

As we prove in what follows, this allows the correct fixing process that we aim for, but this has the cost of having too many nodes become dirty. However, the crucial point here is that not all nodes that become dirty in items (4) and (5) will actually utilize their timestamp

– some will drop their timestamp before competing for becoming active, and hence we will not need to account for fixing them. That is, we add the following item:

(6) When an active node  $v$  changes its label to `true`, all of its dirty neighbors become clean.

**Correctness.** The correctness follows the exact line of proof of the algorithm in Theorem 2, with the modification that making some neighbors dirty in item (5) compensates for not being able to assign them directly with good new labels. That is, at the end of the epoch, we still have the following guarantee: if all nodes are clean, then their labels induce an MIS; otherwise, for every two clean neighbors, either exactly one of them is in the MIS, or both have a neighbor in the MIS.

**Amortized round complexity.** The proof follows the same lines as the previous complexity proofs, with the addition of an accounting argument. This is used to prove that the cumulative number of epochs in which any node becomes active is at most twice the number of topology changes. This proves our claim of an amortized  $O(1)$  round complexity.

First, as in the former algorithms, we note that  $\text{incorrect}(i) \leq \gamma \cdot \text{epochs}(i)$ , and at each epoch at least one timestamp is handled. Thus, we only need to upper bound the number of timestamps created by round  $i$  as a function of  $\text{changes}(i)$ . However, here we need to be much more careful and we can not simply account each change for two timestamps, as some changes create much more timestamps than others.

Consider a node  $v$  that is deleted in round  $i$  as in item (4) (or  $v$  is active and marked `false` as in item (5)), and a set  $Z = \{z_1, \dots, z_k\}$  of its neighbors that become dirty by satisfying the condition in item (4) (or item (5)) above, ordered by their timestamps  $(i, v, z_j)$  for  $1 \leq j \leq k$ , as induced by this topology change. For each  $1 \leq j \leq k$ , if a node  $z_j$  becomes active due to this timestamp, then by item (6), starting from round  $i$  none of its neighbors change their label to `true`. Consider the last round  $i'$  before round  $i$  in which the label of  $z_j$  is `true` ( $i'$  exists since this condition occurs initially when the graph is empty). We claim that the topology change whose associated active node changed the label of  $z_j$  to `false` in round  $i' + 1$ , is either an insertion of an edge  $\{z_j, u\}$  that satisfies the condition of item (2) with  $ID(z_j) < ID(u)$ , or an insertion of the node  $z_j$  which connects it to at least one node whose label is `true`. The reason for this is that these are the only topology changes which cause  $z_j$  to be assigned the label `false`.

Finally, notice that these topology changes both induce only a single dirty node (thus a single active node and a single epoch), and therefore we can blame  $z_j$  becoming active on the corresponding topology change. This is an injective mapping, as any other node cannot blame these changes (they are changes that made  $z_j$  dirty), and  $z_j$  itself may become active again in the future due to satisfying the condition in item (4) (or item (5)) above only if its label is changed to `false` again in between.

In other words, this blaming argument implies  $\text{epochs}(i) \leq 2 \cdot \text{changes}(i)$  here as well, completing the proof.  $\blacktriangleleft$

---

## References

- 1 Sepehr Assadi, Krzysztof Onak, Baruch Schieber, and Shay Solomon. Fully dynamic maximal independent set with sublinear update time. In *STOC*, 2018.
- 2 Sepehr Assadi, Krzysztof Onak, Baruch Schieber, and Shay Solomon. Fully dynamic maximal independent set with sublinear in  $n$  update time. In *SODA*, 2019.
- 3 Hagit Attiya, Danny Dolev, and Nir Shavit. Bounded polynomial randomized consensus. In *PODC*, 1989.

- 4 Yossi Azar, Shay Kutten, and Boaz Patt-Shamir. Distributed error confinement. *ACM Trans. Algorithms*, 2010.
- 5 Alkida Balliu, Sebastian Brandt, Dennis Olivetti, and Jukka Suomela. How much does randomness help with locally checkable problems? *CoRR*, abs/1902.06803, 2019.
- 6 Alkida Balliu, Juho Hirvonen, Janne H. Korhonen, Tuomo Lempiäinen, Dennis Olivetti, and Jukka Suomela. New classes of distributed time complexity. In *STOC*, 2018.
- 7 Philipp Bamberger, Fabian Kuhn, and Yannic Maus. Local distributed algorithms in highly dynamic networks. In *IPDPS*, 2019.
- 8 Leonid Barenboim and Michael Elkin. *Distributed Graph Coloring: Fundamentals and Recent Developments*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2013.
- 9 Leonid Barenboim, Michael Elkin, and Uri Goldenberg. Locally-iterative distributed  $(\Delta+1)$ -coloring below szegedy-vishwanathan barrier, and applications to self-stabilization and to restricted-bandwidth models. In *PODC*, 2018.
- 10 Soheil Behnezhad, Mahsa Derakhshan, Mohammadtaghi Hajiaghayi, Cliff Stein, and Madhu Sudan. Fully dynamic maximal independent set with polylogarithmic update time. *FOCS*, 2019.
- 11 Sayan Bhattacharya, Deeparnab Chakrabarty, Monika Henzinger, and Danupon Nanongkai. Dynamic algorithms for graph coloring. In *SODA*, 2018.
- 12 Matthias Bonne and Keren Censor-Hillel. Distributed detection of cliques in dynamic networks. In *ICALP*, 2019.
- 13 Sebastian Brandt, Orr Fischer, Juho Hirvonen, Barbara Keller, Tuomo Lempiäinen, Joel Rybicki, Jukka Suomela, and Jara Uitto. A lower bound for the distributed lovász local lemma. In *STOC*, 2016.
- 14 Sebastian Brandt, Juho Hirvonen, Janne H. Korhonen, Tuomo Lempiäinen, Patric R. J. Östergård, Christopher Purcell, Joel Rybicki, Jukka Suomela, and Przemyslaw Uznanski. LCL problems on grids. In *PODC*, 2017.
- 15 Keren Censor-Hillel, Elad Haramaty, and Zohar S. Karnin. Optimal dynamic distributed MIS. In *PODC*, 2016.
- 16 Keren Censor-Hillel, Victor I. Kolobov, and Gregory Schwartzman. Finding subgraphs in highly dynamic networks. In *submission*, 2020.
- 17 Yi-Jun Chang, Tsvi Kopelowitz, and Seth Pettie. An exponential separation between randomized and deterministic complexity in the LOCAL model. *SIAM J. Comput.*, 2019.
- 18 Yi-Jun Chang and Seth Pettie. A time hierarchy theorem for the LOCAL model. *SIAM J. Comput.*, 2019.
- 19 Shiri Chechik and Tianyi Zhang. Fully dynamic maximal independent set in expected poly-log update time. *FOCS*, 2019.
- 20 Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000.
- 21 Yuhao Du and Hengjie Zhang. Improved algorithms for fully dynamic maximal independent set. *CoRR*, abs/1804.08908, 2018. URL: <http://arxiv.org/abs/1804.08908>.
- 22 Michael Elkin. A near-optimal distributed fully dynamic algorithm for maintaining sparse spanners. In *PODC*, 2007.
- 23 Mohsen Ghaffari, Fabian Kuhn, and Yannic Maus. On the complexity of local distributed graph problems. In *STOC*, 2017.
- 24 Nabil Guellati and Hamamache Kheddouci. A survey on self-stabilizing algorithms for independence, domination, coloring, and matching in graphs. *J. Parallel Distrib. Comput.*, 2010.
- 25 Manoj Gupta and Shahbaz Khan. Simple dynamic algorithms for maximal independent set and other problems. *CoRR*, abs/1804.01823, 2018. [arXiv:1804.01823](https://arxiv.org/abs/1804.01823).
- 26 Giuseppe F. Italiano. Distributed algorithms for updating shortest paths (extended abstract). In *WDAG*, 1991.

## 28:16 Fast Deterministic Algorithms for Highly-Dynamic Networks

- 27 Ken-ichi Kawarabayashi and Gregory Schwartzman. Adapting local sequential algorithms to the distributed setting. In *DISC*, 2018.
- 28 Michael König and Roger Wattenhofer. On local fixing. In *OPODIS*, 2013.
- 29 Fabian Kuhn, Nancy A. Lynch, and Rotem Oshman. Distributed computation in dynamic networks. In *STOC*, 2010.
- 30 Shay Kutten and David Peleg. Fault-local distributed mending. *J. Algorithms*, 1999.
- 31 Shay Kutten and David Peleg. Tight fault locality. *SIAM J. Comput.*, 2000.
- 32 Moni Naor and Larry J. Stockmeyer. What can be computed locally? *SIAM J. Comput.*, 1995.
- 33 Merav Parter, David Peleg, and Shay Solomon. Local-on-average distributed tasks. In *SODA*, 2016.
- 34 Shay Solomon. Fully dynamic maximal matching in constant update time. In *FOCS*, 2016.
- 35 Jukka Suomela. Survey of local algorithms. *ACM Comput. Surv.*, 2013.



# Approximating Bipartite Minimum Vertex Cover in the CONGEST Model

Salwa Faour

Albert-Ludwigs-Universität Freiburg, Germany  
salwa.faour@cs.uni-freiburg.de

Fabian Kuhn

Albert-Ludwigs-Universität Freiburg, Germany  
kuhn@cs.uni-freiburg.de

---

## Abstract

We give efficient distributed algorithms for the minimum vertex cover problem in bipartite graphs in the CONGEST model. From König's theorem, it is well known that in bipartite graphs the size of a minimum vertex cover is equal to the size of a maximum matching. We first show that together with an existing  $O(n \log n)$ -round algorithm for computing a maximum matching, the constructive proof of König's theorem directly leads to a deterministic  $O(n \log n)$ -round CONGEST algorithm for computing a minimum vertex cover. We then show that by adapting the construction, we can also convert an *approximate* maximum matching into an *approximate* minimum vertex cover. Given a  $(1 - \delta)$ -approximate matching for some  $\delta > 1$ , we show that a  $(1 + O(\delta))$ -approximate vertex cover can be computed in time  $O(D + \text{poly}(\frac{\log n}{\delta}))$ , where  $D$  is the diameter of the graph. When combining with known graph clustering techniques, for any  $\varepsilon \in (0, 1]$ , this leads to a  $\text{poly}(\frac{\log n}{\varepsilon})$ -time deterministic and also to a slightly faster and simpler randomized  $O(\frac{\log n}{\varepsilon^3})$ -round CONGEST algorithm for computing a  $(1 + \varepsilon)$ -approximate vertex cover in bipartite graphs. For constant  $\varepsilon$ , the randomized time complexity matches the  $\Omega(\log n)$  lower bound for computing a  $(1 + \varepsilon)$ -approximate vertex cover in bipartite graphs even in the LOCAL model. Our results are also in contrast to the situation in general graphs, where it is known that computing an optimal vertex cover requires  $\tilde{\Omega}(n^2)$  rounds in the CONGEST model and where it is not even known how to compute any  $(2 - \varepsilon)$ -approximation in time  $o(n^2)$ .

**2012 ACM Subject Classification** Theory of computation → Distributed algorithms; Networks → Network algorithms

**Keywords and phrases** distributed vertex cover, distributed graph algorithms, distributed optimization, bipartite vertex cover

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2020.29

**Related Version** A full version of the paper is available at <https://arxiv.org/abs/2011.10014>.

## 1 Introduction & Related Work

In the minimum vertex cover (MVC) problem, we are given an  $n$ -node graph  $G = (V, E)$  and we are asked to find a vertex cover of smallest possible size, that is, a minimum cardinality subset of  $V$  that contains at least one node of every edge in  $E$ . In the distributed MVC problem, the graph  $G$  is the network graph and the nodes of  $G$  have to compute a vertex cover by communicating over the edges of  $G$ . At the end of a distributed vertex cover algorithm, every node  $v \in V$  must know if it is contained in the vertex cover or not. Different variants of the MVC problem have been studied extensively in the distributed setting, see e.g., [3, 4, 6, 7, 9, 11, 16, 18–20, 25, 26]. Classically, when studying the distributed MVC problem and also related distributed optimization problems on graphs, the focus has been on understanding the *locality* of the problem. The focus therefore has mostly been



© Salwa Faour and Fabian Kuhn;  
licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 29; pp. 29:1–29:16

Leibniz International Proceedings in Informatics



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

on establishing how many synchronous communication rounds are necessary to solve or approximate the problem in the LOCAL model, that is, if in each round, each node of  $G$  can send an arbitrarily large message to each of its neighbors.

**MVC in the LOCAL model.** The minimum vertex cover problem is closely related to the maximum matching problem, i.e., to the problem of finding a maximum cardinality set of pairwise non-adjacent (i.e., disjoint) edges. Since for every matching  $M$ , any vertex cover has to contain at least one node from each of the edges  $\{u, v\} \in M$ , the size of a minimum vertex cover is lower bounded by the size of a maximum matching. We therefore obtain a simple 2-approximation  $S$  for the MVC problem by first computing a maximal matching and by defining the vertex cover  $S$  as  $S := \bigcup_{\{u,v\} \in M} \{u, v\}$ . It has been known since the 1980s that a maximal matching can be computed in  $O(\log n)$  rounds by using a simple randomized algorithm [2, 22, 29]. The fastest known randomized distributed algorithm for computing a maximal matching has a round complexity of  $O(\log \Delta + \log^3 \log n)$ , where  $\Delta$  is the maximum degree of the graph  $G$  [8, 15], and the fastest known deterministic algorithm has a round complexity of  $O(\log^2 \Delta \cdot \log n)$  [15]. A slightly worse approximation ratio of  $2 + \varepsilon$  can even be achieved in time  $O\left(\frac{\log \Delta}{\log \log \Delta}\right)$  for any constant  $\varepsilon > 0$ . This matches the  $\Omega\left(\min\left\{\frac{\log \Delta}{\log \log \Delta}, \sqrt{\frac{\log n}{\log \log n}}\right\}\right)$  lower bound of [25], which even holds for any polylogarithmic approximation ratio. In [18], it was further shown that there exists a constant  $\varepsilon > 0$  such that computing a  $(1 + \varepsilon)$ -approximate solution for MVC requires  $\Omega(\log n)$  rounds even for bipartite graphs of maximum degree 3. By using known randomized distributed graph clustering techniques [27, 30], this bound can be matched: For any  $\varepsilon \in (0, 1]$ , a  $(1 + \varepsilon)$ -approximate MVC solution can be computed in time  $O\left(\frac{\log n}{\varepsilon}\right)$  in the LOCAL model. It was shown in [17] that in fact all distributed covering and packing problems can be  $(1 + \varepsilon)$ -approximated in time  $\text{poly}\left(\frac{\log n}{\varepsilon}\right)$  in the LOCAL model. By combining with the recent deterministic network decomposition algorithm of [32], the same result can even be achieved deterministically. We note that all the distributed  $(1 + \varepsilon)$ -approximations for MVC and related problems quite heavily exploit the power of the LOCAL model. They use very large messages and also the fact that the nodes can do arbitrary (even exponential-time) computations for free.

**MVC in the CONGEST model.** As the complexity of the distributed minimum vertex cover and related problems in the LOCAL model is now understood quite well, there has recently been increased interest in also understanding the complexity of these problems in the more restrictive CONGEST model, that is, when assuming that in each round, every node can only send an  $O(\log n)$ -bit message to each of its neighbors. Some of the algorithms that have been developed for the LOCAL model do not make use of large messages and they therefore directly also work in the CONGEST model. This is in particular true for all the maximal matching algorithms and also for the  $(2 + \varepsilon)$ -approximate MVC algorithm mentioned above. Also in the CONGEST model, it is therefore possible to compute a 2-approximation for MVC in  $O(\log \Delta + \log^3 \log n)$  rounds and a  $(2 + \varepsilon)$ -approximation in  $O\left(\frac{\log \Delta}{\log \log \Delta}\right)$  rounds. However, there is no non-trivial (i.e.,  $o(n^2)$ -round) CONGEST MVC algorithm known for obtaining an approximation ratio below 2. For computing an optimal vertex cover on general graphs, it is even known that  $\tilde{\Omega}(n^2)$  rounds are necessary in the CONGEST model [11]. It is therefore an interesting open question to investigate if it is possible to approximate MVC within a factor smaller than 2 in the CONGEST model or to understand for which families of graphs, this is possible. The only result in this direction that we are aware of is a recent paper that gives  $(1 + \varepsilon)$ -approximation for MVC in the square graph  $G^2$  in  $O(n/\varepsilon)$  CONGEST rounds on the underlying graph  $G$  [6].

**MVC in bipartite graphs.** In the present paper, we study the distributed complexity of MVC in the CONGEST model for bipartite graphs. Unlike for general graphs, where MVC is APX-hard (and even hard to approximate within a factor  $2 - \varepsilon$  when assuming the unique games conjecture [24]), for bipartite graphs, MVC can be solved optimally in polynomial time. While in general graphs, we only know that a minimum vertex cover is at least as large as a maximum matching and at most twice as large as a maximum matching, for bipartite graphs, König's well-known theorem [12, 23] states that in bipartite graphs, the size of a maximum matching is always equal to the size of a minimum vertex cover. In fact, if one is given a maximum matching of a bipartite graph  $G = (U \cup V, E)$ , a vertex cover of the same size can be computed in the following simple manner. Assume that we are given the bipartition of the nodes of  $G$  into sets  $U$  and  $V$  and assume that we are given a maximum matching  $M$  of  $G$ . Now, let  $L_0 \subseteq U$  be the set of unmatched nodes in  $U$  and let  $L \subseteq U \cup V$  be the set of nodes that are reachable from  $L_0$  over an alternating path (i.e., over a path that alternates between edges in  $E \setminus M$  and edges in  $M$ ). It is not hard to show that the set  $S := (U \setminus L) \cup (V \cap L)$  is a vertex cover that contains exactly one node of every edge in  $M$ . We note that this construction also directly leads to a distributed algorithm for computing an optimal vertex cover in bipartite graphs  $G$ . The bipartition of  $G$  can clearly be computed in time  $O(D)$ , where  $D$  is the diameter of  $G$  and given a maximum matching  $M$ , the set  $L$  can then be computed in  $O(n)$  rounds by doing a parallel BFS exploration on alternating paths starting at all nodes in  $L_0$ . Together with the  $O(n \log n)$ -round CONGEST algorithm of [1] for computing a maximum matching, this directly leads to a deterministic  $O(n \log n)$ -round CONGEST algorithm for computing an optimal vertex cover in bipartite graphs. As our main contribution, we show that it is not only possible to efficiently convert an optimal matching into an optimal vertex cover, but we can also efficiently turn an *approximate* solution of the maximum matching problem in a bipartite graph into an *approximate* solution of the MVC problem on the same graph. Unlike for MVC, where no arbitrarily good approximation algorithms are known for the CONGEST model, such algorithms are known for the maximum matching problem [1, 5, 28]. We use this to develop polylogarithmic-time approximation schemes for the bipartite MVC problem in the CONGEST model. We next discuss our main contributions in more detail.

## 1.1 Contributions

Our first contribution is a simple linear-time algorithm to solve the exact minimum vertex cover problem.

► **Theorem 1.** *There is a deterministic CONGEST algorithm to (exactly) solve the minimum vertex cover problem in bipartite graphs in time  $O(\text{OPT} \cdot \log \text{OPT})$ , where  $\text{OPT}$  is the size of a minimum vertex cover.*

**Proof.** As mentioned, the algorithm is a straightforward CONGEST implementation of König's constructive proof. Given a bipartite graph  $G = (U \cup V, E)$ , one first computes a maximum matching  $M$  of  $G$  in time  $O(\text{OPT} \cdot \log \text{OPT})$  by using the CONGEST algorithm of [1]. One elects a leader node  $\ell$  and computes a BFS tree of  $G$  rooted at  $\ell$  in time  $O(D)$ , where  $D$  is the diameter of  $G$ . Let  $U$  be the set of nodes at even distance from  $\ell$  and let  $V$  be the set of nodes at odd distance from  $\ell$ . Let  $L_0$  be the set of nodes in  $U$  that are not contained in any edge of  $M$ . Starting at  $L_0$ , we do a parallel BFS traversal on alternating paths. Let  $L$  be the set of nodes that are reached in this way. The set  $L$  can clearly be computed in time  $O(|M|) = O(\text{OPT})$ . As shown in the constructive proof of König's theorem [12, 23], the minimum vertex cover  $S$  is now defined as  $S := (U \setminus L) \cup (V \cap L)$ . ◀

Our main results are two distributed algorithms to efficiently compute  $(1 + \varepsilon)$ -approximate solutions to the minimum vertex cover problem. We first give a slightly more efficient (and also somewhat simpler) randomized algorithm.

► **Theorem 2.** *For every  $\varepsilon \in (0, 1]$ , there is a randomized CONGEST algorithm that for any bipartite  $n$ -node graph  $G$  computes a vertex cover of expected size at most  $(1 + \varepsilon) \cdot \text{OPT}$  in time  $O\left(\frac{\log n}{\varepsilon^3}\right)$ , w.h.p., where  $\text{OPT}$  is the size of a minimum vertex cover of  $G$ .*

We remark that for constant  $\varepsilon$ , the above result matches the lower bound of [18] for the LOCAL model. More precisely, in [18], it is shown that there exists a constant  $\varepsilon > 0$  for which computing a  $(1 + \varepsilon)$ -approximation of minimum vertex cover requires  $\Omega(\log n)$  rounds even on bounded-degree bipartite graphs. The second main result shows that similar bounds can also be achieved deterministically.

► **Theorem 3.** *For every  $\varepsilon \in (0, 1]$ , there is a deterministic CONGEST algorithm that for any bipartite  $n$ -node graph  $G$  computes a vertex cover of size at most  $(1 + \varepsilon) \cdot \text{OPT}$  in time  $\text{poly}\left(\frac{\log n}{\varepsilon}\right)$ , where  $\text{OPT}$  is the size of a minimum vertex cover of  $G$ .*

## 1.2 Our Techniques in a Nutshell

We next describe the key ideas that leads to the results in Theorems 2 and 3. The core of our algorithms is a method to efficiently transform an approximate solution  $M$  for the maximum matching problem into an approximate solution of MVC. More concretely, assume that we are given a matching  $M \subseteq E$  of a bipartite graph  $G = (U \cup V, E)$  such that  $M$  is a  $(1 - \varepsilon)$ -approximate maximum matching of  $G$  (for a sufficiently small  $\varepsilon > 0$ ). In Section 3, we then first show that we can compute a vertex cover  $S \subseteq U \cup V$  of size  $(1 + O(\varepsilon \text{ poly } \log n)) \cdot |M|$  (and therefore a  $(1 + O(\varepsilon \text{ poly } \log n))$ -approximation for MVC) in time  $O(D + \text{poly}\left(\frac{\log n}{\varepsilon}\right))$ , where  $D$  is the diameter of  $G$ . If the matching  $M$  has the additional property that there are no augmenting paths of length at most  $2k - 1$  for some  $k = O(1/\varepsilon)$ , we show that such a vertex cover  $S$  can be obtained by adapting the constructive proof of König's theorem. Clearly, the bipartition of the nodes of  $G$  into sets  $U$  and  $V$  can be computed in time  $O(D)$ . Now, we again define  $L_0$  as the set of unmatched nodes in  $U$  and more generally for any integer  $i \in \{1, \dots, 2k\}$ , we define  $L_i$  to be the set of nodes in  $U \cup V$  that can be reached over an alternating path of length  $i$  from  $L_0$  and for which no shorter such alternating path exists. Note that all nodes in set  $L_{2j-1}$  for  $j \in \{1, \dots, k\}$  are matched nodes as otherwise, we would have an augmenting path of length at most  $2k - 1$ . Note that any alternating path starting at  $L_0$  starts with a non-matching edge from  $U$  to  $V$  and it alternates between non-matching edges from  $U$  to  $V$  and matching edges from  $V$  to  $U$ . For every  $j \geq 1$ , the set  $L_{2j}$  therefore exactly contains the matching neighbors of the nodes in  $L_{2j-1}$  and we therefore have  $|L_{2j}| = |L_{2j-1}|$ . We will show that for every  $j \in \{1, \dots, k\}$  the set

$$S_j := \bigcup_{j' \in \{1, \dots, j\}} L_{2j'-1} \cup \left( U \setminus \bigcup_{j' \in \{0, \dots, j-1\}} L_{2j'} \right)$$

is a vertex cover of size  $|M| + |L_{2j}| = |M| + |L_{2j-1}|$ . Because the sets  $L_i$  are disjoint, clearly one of these vertex covers must have size at most  $(1 + \frac{1}{k}) \cdot |M| = (1 + O(\varepsilon)) \cdot |M|$ .

If we do not have the guarantee that  $M$  does not have short augmenting paths, we show that one can first delete  $O(\varepsilon \cdot |M| \cdot \text{poly } \log n)$  nodes from  $U \cup V$  such that in the induced subgraph of the remaining nodes, there are no short augmenting paths w.r.t.  $M$ . We also

show that we can find such a set of nodes to delete in time  $\text{poly}\left(\frac{\log n}{\varepsilon}\right)$ . We can therefore then first compute a good vertex cover approximation for the remaining graph and we then obtain a vertex cover of  $G$  by also adding all the removed nodes to the vertex cover.

Given our algorithm to compute a good MVC approximation in time  $O(D + \text{poly log } n)$  in Section 4, we show how that in combination with known graph clustering techniques, we can obtain MVC approximation algorithms with polylogarithmic time complexities and thus prove Theorems 2 and 3. Given a maximal matching  $M$ , we show that we can compute disjoint low-diameter clusters such that all the edges between clusters can be covered by  $O(\varepsilon \cdot |M|)$  nodes. With randomization, such a clustering can be computed by using the random shifts approach of [10, 30] and deterministically such a clustering can be computed by a simple adaptation of the recent network decomposition algorithm of [32]. Since the clusters have a small diameter, we can then use the algorithm of Section 3 described above inside the clusters to efficiently compute a good MVC approximation.

## 2 Model and Definitions

**Communication Model.** We work with the standard CONGEST model [31]. The network is modelled as an  $n$ -node undirected graph  $G = (V, E)$  with maximum degree at most  $\Delta$  and each node has a unique  $O(\log n)$ -bit identifier. The computation proceeds in synchronous communication rounds. Per round, each node can perform some local computations and send one  $O(\log n)$ -bit message to each of its neighbors. At the end, each node should know its own part of the output, e.g., whether it belongs to a vertex cover or not.

**Low-Diameter Clustering.** In order to reduce the problem of approximating MVC on general (bipartite) graphs to approximating MVC on low-diameter (bipartite) graphs, we need a slightly generalized form of a standard type of graph clustering. Let  $G = (V, E, w)$  be a weighted graph with non-negative edge weights  $w(e)$  and assume that  $W := \sum_{w \in E} w(e)$  is the total weight of all edges in  $G$ . A subset  $S \subseteq V$  of the nodes of  $G$  is called  $\lambda$ -dense for  $\lambda \in [0, 1]$  if the total weight of the edges of the induced subgraph  $G[S]$  is at least  $\lambda \cdot W$ . A *clustering* of  $G$  is a collection  $\{S_1, \dots, S_k\}$  of disjoint subsets  $S_i \subseteq V$  of the nodes. A clustering  $\{S_1, \dots, S_k\}$  is called  $\lambda$ -dense if the set  $S := S_1 \cup \dots \cup S_k$  is  $\lambda$ -dense. The *strong diameter* of a cluster  $S_i \subseteq V$  is the (unweighted) diameter of the induced subgraph  $G[S_i]$  and the *weak diameter* of a cluster  $S_i \subseteq V$  is the maximum (unweighted) distance in  $G$  between any two nodes in  $S_i$ . The strong/weak diameter of a clustering  $\{S_1, \dots, S_k\}$  is the maximum strong/weak diameter of any cluster  $S_i$ . A clustering  $\{S_1, \dots, S_k\}$  is called  *$h$ -hop separated* for some integer  $h \geq 1$  if for any two clusters  $S_i$  and  $S_j$  ( $i \neq j$ ), we have  $\min_{(u,v) \in S_i \times S_j} d_G(u,v) \geq h$ , where  $d_G(u,v)$  denotes the hop-distance between  $u$  and  $v$  in  $G$ . A clustering  $\{S_1, \dots, S_k\}$  is called  *$(c, d)$ -routable* if we are in addition given a collection of trees  $T_1, \dots, T_k$  in  $G$  such that for every  $i \in \{1, \dots, k\}$ , the node set of  $T_i$  contains the nodes in  $S_i$ , the height of  $T_i$  is at most  $d$  and every edge  $e \in E$  of  $G$  is contained in at most  $c$  trees  $T_1, \dots, T_k$ . Note that a  $(c, d)$ -routable clustering clearly has weak diameter at most  $2d$ . Note also that any clustering with strong diameter  $d$  can easily be extended to a  $(1, d)$ -routable clustering by computing a BFS tree  $T_i$  for the induced subgraph  $G[S_i]$  of each cluster  $S_i$ .

## 3 Approximating MVC in Time Linear in the Diameter

In this section, we show how to compute a minimum vertex cover approximation in time  $O(D + \text{poly log } n)$  in the CONGEST model, where  $D$  is the diameter of the graph. Before discussing a distributed algorithm, we first describe a generic high-level algorithm to compute

a  $(1 - \varepsilon)$ -approximate vertex cover from an appropriate approximate matching  $M$  of a bipartite graph  $G$ . Given a matching  $M$  of any graph  $G$ , a path is said to be augmenting w.r.t.  $M$  in  $G$  if it is a path that starts and ends with unmatched vertices and alternates between matched and unmatched edges. Inspired by the standard constructive proof of König's theorem, we first describe an algorithm that gives an approximate minimum vertex cover in bipartite graphs from an approximate maximum matching with the guarantee that no short augmenting paths exist in the graph. We remark that a similar construction has also been used by Feige, Mansour, and Schapire for approximating the bipartite MVC problem in the local computation algorithms model [14].

In the following, assume that  $G = (V, E)$  is a bipartite graph, where the bipartition of  $V$  is given by  $V = A \cup B$ . Let  $k \geq 1$  be an integer parameter and assume that  $M$  is a matching of  $G$  with no augmenting paths of length  $2k - 1$  or shorter. We further define a directed version  $\vec{G}$  of the graph  $G$ , where every edge  $e \notin M$  is directed from set  $A$  to set  $B$  and every edge  $e \in M$  is directed from  $B$  to  $A$  (note that by definition of  $A$  and  $B$ , every edge of  $G$  is between a node in  $A$  and a node in  $B$ ). We then apply the following algorithm to compute a set  $S$ , which we will show is a  $(1 - 1/k)$ -approximate vertex cover of  $G$ .

#### Basic Approximate Vertex Cover Algorithm

1. Let  $A_0 \subseteq A$  be the set of unmatched nodes in  $A$ .
2. For every  $i \in \{1, \dots, k\}$ , let  $A_i \subseteq A$  be the set of nodes in  $A$  for which the shortest directed path in  $\vec{G}$  from a node in  $A_0$  is of length  $2i$ .
3. For every  $i \in \{1, \dots, k\}$ , let  $B_i \subseteq B$  be the set of nodes in  $\vec{G}$  from a nodes in  $A_0$  is of length  $2i - 1$ .
4. Define  $i^* := \arg \min_{i \in \{1, \dots, k\}} |B_i|$ .
5. Output  $S := \bigcup_{i=1}^{i^*} B_i \cup (A \setminus \bigcup_{i=0}^{i^*-1} A_i)$ .

► **Lemma 4.** *If the given matching  $M$  has no augmenting paths of length at most  $2k - 1$ , the above algorithm computes a vertex cover  $S$  of  $G$  of size at most  $(1 + 1/k) \cdot \text{OPT}$ , where  $\text{OPT}$  is the size of a minimum vertex cover of  $G$ .*

**Proof.** We first show that  $S$  is a vertex cover of  $G$ . A bit more generally, for any  $\hat{i} \in \{1, \dots, k\}$ , we define  $S_{\hat{i}} := \bigcup_{i=1}^{\hat{i}} B_i \cup (A \setminus \bigcup_{i=0}^{\hat{i}-1} A_i)$  and show that  $S_{\hat{i}}$  is a vertex cover of  $G$ . For  $S_{\hat{i}}$  to not be a vertex cover, there must be an a node  $u$  in a set  $A_j$  for  $j \in \{0, \dots, \hat{i} - 1\}$  and a node  $v$  in  $B \setminus \bigcup_{j=1}^{\hat{i}} B_j$ . Note that the edge  $\{u, v\}$  cannot be a matching edge because either  $u \in A_0$ , in which case  $u$  is unmatched, or  $u \in A_j$  for  $j < \hat{i}$ . In the second case,  $u$  is reached over a path of length  $2j$  in the directed graph  $\vec{G}$  and thus  $u$ 's matching edge connects to a node in  $B_j$ . However, if  $\{u, v\}$  is not a matching edge, it means that the edge  $\{u, v\}$  is directed from  $u$  to  $v$  in graph  $\vec{G}$ . Therefore, since the shortest directed path from  $A_0$  to  $u$  in  $\vec{G}$  is of length  $2j$ , there must be a directed path from  $A_0$  to  $v$  of length at most  $2j + 1$  in  $\vec{G}$ . This means that  $v$  must be in one of the sets  $B_1, \dots, B_{j+1}$ . Since  $j < \hat{i}$ , this contradicts the assumption that  $v \in B \setminus \bigcup_{j=1}^{\hat{i}} B_j$ . We can therefore conclude that  $S_{\hat{i}}$  is a vertex cover of  $G$  for every  $\hat{i} \in \{1, \dots, k\}$  and thus, in particular, the set  $S_{i^*}$  is a vertex cover of  $G$ .

It remains to show that the size of  $S$  is at most  $(1 + 1/k) \cdot \text{OPT}$ . To prove this, we first show that for all  $i \in \{1, \dots, k\}$ , the set of nodes in  $B_i$  are matched nodes. If there is a node  $v \in B_i$  for  $i \leq k$  that is unmatched, there is a directed path of length  $2i - 1 \leq 2k - 1$  in  $\vec{G}$  from a node in  $A_0$  to  $v$ . Such a directed path would be an augmenting path of the same length  $2i - 1 \leq 2k - 1$  w.r.t. matching  $M$  in  $G$ . This cannot be because we assumed that there are no augmenting paths of length at most  $2k - 1$  w.r.t.  $M$  in  $G$ . Further, by definition



of the directed graph  $\vec{G}$ , the reason that a node  $u$  is in a set  $A_i$  for  $i \in \{1, \dots, k\}$  is that the matching edge of  $u$  connects  $u$  to a node in  $B_i$ . By induction on  $i$ , we can therefore conclude that for all  $i \in \{1, \dots, k\}$ , the nodes in  $A_i$  are exactly the matching neighbors of the nodes in  $B_i$  and therefore for all such  $i$ , we have  $|A_i| = |B_i|$ . For every  $\hat{i} \in \{1, \dots, k\}$ , the size of the set  $S_{\hat{i}}$  can therefore be computed as

$$|S_{\hat{i}}| = \sum_{i=1}^{\hat{i}} |B_i| + \underbrace{|A| - |A_0|}_{=|M|} - \sum_{i=1}^{\hat{i}-1} |A_i| = |M| + |B_{\hat{i}}|.$$

Because the sets  $B_i$  for  $i \in \{1, \dots, k\}$  are disjoint and they all contain matched nodes, their total size is at most  $|M|$  and therefore, we have  $|B_{i^*}| \leq |M|/k$ . We can therefore conclude that  $|S| = |M| + |B_{i^*}| \leq (1 + 1/k) \cdot |M| = (1 + 1/k) \cdot \text{OPT}$ . ◀

We next discuss how the above algorithm can efficiently be implemented in time  $O(D + \text{poly log } n)$  in the CONGEST model, where  $D$  is the diameter of the graph. A bit more precisely, we will show the following. Let  $G = (V, E)$  be a bipartite graph with diameter  $D$  and let  $G' = (V', E')$  be a subgraph of  $G$ . Assume that each node of  $G$  knows if it is contained in the set  $V'$  and which of its edges are contained in the set  $E'$ . We then show that for any  $k \geq 1$ , one can run the above algorithm on graph  $G'$  in  $O(D + k)$  rounds in the CONGEST model on graph  $G$ . The implementation is relatively straightforward. In time  $O(D)$ , one can compute a BFS tree of the graph  $G$ , and one can compute the bipartition of the nodes into sets  $A$  and  $B$ . Then, in  $O(k)$  rounds, one can do the BFS traversal on the directed graph  $\vec{G}$ , starting from nodes in  $A_0$  and computing the sets  $A_i$  and  $B_i$  for  $i \in \{1, \dots, k\}$ . Finally, by using the BFS tree on graph  $G$  and a simple pipelining scheme, one can compute the sizes of all the sets  $B_i$  and determine the index  $i^*$  of the smallest such set. A formal statement is given by the following lemma and a formal proof of the lemma can be found in the full version of the paper [13].

► **Lemma 5.** *Let  $G = (V, E)$  be a bipartite graph of diameter  $D$ , let  $G' = (V', E')$  be a subgraph of  $G$  (i.e.,  $V' \subseteq V$  and  $E' \subseteq E$ ), and let  $k \geq 1$  be an integer parameter. Assume that  $M$  is a matching of  $G'$  s.t. there exists no augmenting path of length at most  $2k - 1$  w.r.t.  $M$  in  $G'$ . Then, there exists a deterministic CONGEST model algorithm to compute a  $(1 + 1/k)$ -approximate minimum vertex cover of  $G'$  in  $O(D + k)$  rounds on graph  $G$ .*

In combination with a distributed approximate maximum matching algorithm of Lotker, Patt-Shamir, and Pettie [28], Lemma 5 directly leads to a randomized  $O(D + \text{poly log } n)$ -round distributed approximation scheme for the MVC problem.

► **Theorem 6.** *Let  $G = (V, E)$  be a bipartite graph of diameter  $D$  and  $G' = (V', E')$  be a subgraph of  $G$  (i.e.,  $V' \subseteq V$  and  $E' \subseteq E$ ). For  $\varepsilon \in (0, 1]$ , there is a randomized algorithm that gives a  $(1 + \varepsilon)$ -approximate minimum vertex cover of  $G'$  w.h.p. in  $O(D + \frac{\log n}{\varepsilon^3})$  rounds in the CONGEST model on  $G$ .*

**Proof.** The approximate maximum matching algorithm of [28] is based on the classic approach of Hopcroft and Karp [21]. For a given graph and positive integer parameter  $k$ , the algorithm computes a matching  $M$  of the graph such that there is no augmenting path of length at most  $2k - 1$  w.r.t.  $M$ . When run on an  $n$ -node graph, the algorithm w.h.p. has a time complexity of  $O(k^3 \cdot \log n)$  in the CONGEST model. The theorem therefore directly follows by applying the algorithm of [28] on  $G'$  with  $k = \lceil 1/\varepsilon \rceil$  and by Lemma 5. ◀



### 3.1 Deterministic MVC Approximation

The only part in the algorithm underlying Theorem 6 that is randomized is the approximate maximum matching algorithm of [28]. In order to also obtain a deterministic distributed MVC algorithm, we therefore have to replace the randomized distributed matching algorithm by a deterministic distributed matching algorithm. The algorithm of [28] is based on the framework of [21] and it therefore guarantees that the resulting matching has no short augmenting paths. While the size of such a matching is guaranteed to be close to the size of a maximum matching, the converse is not necessarily true.<sup>1</sup> Unfortunately, we are not aware of an efficient deterministic CONGEST model algorithm to compute a matching  $M$  with no short augmenting paths. To resolve this issue, we therefore have to do some additional work.

For  $\varepsilon > 0$ , we define an augmenting path w.r.t. a matching in  $G'$  to be short if it is of length at most  $\ell = 2k' - 1$ , where  $k' = \lceil 2/\varepsilon \rceil$ . We define  $\delta \leq \varepsilon/(2\alpha)$  where  $\alpha = O(\frac{\log \Delta}{\varepsilon^3})$ . We first run a polylogarithmic-time deterministic CONGEST algorithm by Ahmadi et al. [1] to obtain a  $(1 - \delta)$ -approximate maximum matching  $M$  in  $G'$ . This matching  $M$  can potentially have short augmenting paths. In order to get rid of short augmenting paths, we then find a subset of nodes  $S_1$  such that after deleting the nodes in  $S_1$ ,  $M$  is a matching with no short augmenting paths in the remaining subgraph  $G''$  of  $G'$ . We show that we can select  $S_1$  such that  $|S_1| \leq \alpha\delta\text{OPT}$ , where  $\text{OPT}$  is the size of a minimum vertex cover in  $G'$ . Now that we end up with a matching in  $G''$  with no short augmenting paths, we can directly apply our subroutine from above on  $G''$  and obtain a set  $S_2$  which is a  $(1 + \frac{\varepsilon}{2})$ -approximate vertex cover of  $G''$ . Finally, we deduce that  $C = S_1 \cup S_2$  is a vertex cover of  $G'$ . Moreover, since the size of the minimum vertex cover of  $G''$  is at most  $\text{OPT}$ , we get  $|C| = |S_1| + |S_2| \leq \alpha\delta\text{OPT} + (1 + \frac{\varepsilon}{2})\text{OPT} = (1 + \varepsilon)\text{OPT}$ .

**Finding  $S_1$ .** We next describe an algorithm to compute the set  $S_1$ . We assume that we are given an arbitrary  $(1 - \delta)$ -approximate matching  $M$  of  $G' = (U' \cup V', E')$ . As discussed above, we need to find a node set  $S_1 \subseteq U' \cup V'$  that allows to get rid of augmenting paths of length at most  $\ell = 2k' - 1$ . This will be done in  $(\ell + 1)/2$  stages  $d = 1, 3, \dots, \ell$ . The objective of stage  $d$  is to get rid of augmenting paths of length exactly  $d$ . Note that this guarantees that when starting stage  $d$ , there are no augmenting paths of length less than  $d$  and thus in stage  $d$ , all augmenting paths of length  $d$  are also shortest augmenting paths. In the following, we focus on a single stage  $d$ . Formally, the subproblem that we need to solve in stage  $d$  is the following.

We are given a bipartite graph  $H = (U_H \cup V_H, E_H)$  with at most  $n$  nodes and we are given a matching  $M_H$  of  $H$ . We assume that the bipartition of the graph into  $U_H$  and  $V_H$  is given. Let  $d$  be a positive odd integer and assume that  $H$  has no augmenting paths of length shorter than  $d$  w.r.t.  $M_H$ . The goal is to find a set  $S_H \subseteq U_H \cup V_H$  that is as small as possible such that when removing the set  $S_H$  from the nodes of  $H$  and the resulting induced subgraph  $H' := H[U_H \cup V_H \setminus S_H]$  has no augmenting paths of length at most  $d$  w.r.t. the matching  $M'_H := M_H \cap E(H')$ , i.e., w.r.t. to the matching induced by  $M_H$  in the induced subgraph  $H'$  of the remaining nodes.

We therefore need to find a set  $S_H$  of nodes of  $H$  such that  $S_H$  contains at least one node of every augmenting path of length  $d$  w.r.t.  $M_H$  in graph  $H$ . Further, we want to make sure that after removing  $S_H$ , in the remaining induced subgraph  $H'$  w.r.t. the remaining matching

---

<sup>1</sup> One can for example obtain an almost-maximum matching  $M$  for some graph  $G$  by taking a maximum matching of  $G$  and flipping an arbitrary matched edge to unmatched. While the matching  $M$  is obviously a very good approximate matching, it has a short augmenting path of length 1.

$M'_H$ , there are no augmenting paths that were not present in graph  $H$  w.r.t. matching  $M_H$ . To guarantee this, we make sure that whenever we add a matched node in  $U_H \cup V_H$  to  $S_H$ , we also add its matched neighbor to  $S_H$ . In this way, every node that is unmatched in  $H'$  was also unmatched in  $H$  and therefore any augmenting path in  $H'$  is also an augmenting path in  $H$ .

**Getting Rid of Short Augmenting Paths by Solving Set Cover.** The problem of finding a minimal such collection of matching edges and unmatched nodes can be phrased as a minimum set cover problem. The ground set  $\mathcal{P}$  is the set of all augmenting paths of length  $d$  w.r.t.  $M_H$  in  $H$ . For each unmatched node  $v \in U_H \cup V_H$ , we define  $P_v$  as the set of augmenting paths of length  $d$  that contain  $v$ . Similarly, for each matching edge  $e \in M_H$ , we define  $P_e$  as the set of augmenting paths of length  $d$  that contain  $e$ . The goal is to find a smallest set  $C$  consisting of unmatched nodes  $v$  in  $U_H \cup V_H$  and matching edges  $e \in M_H$  such that the union of the corresponding sets  $P_v$  and  $P_e$  of paths covers all paths in  $\mathcal{P}$ . The set  $S_H$  then consists of all nodes in  $C$  and both nodes of each edge in  $C$ . Let us first have a look at the structure of augmenting paths of length  $d$  in  $H$ . Let  $L_0$  be the set of unmatched nodes in  $U_H$  and more generally let  $L_i \subseteq U_H \cup V_H$  for  $i \in \{0, \dots, d\}$  be the set of nodes of  $H$  that can be reached over a shortest alternating path of length  $i$  from a node in  $L_0$ . Since the bipartition into  $U_H$  and  $V_H$  is given, the sets  $L_0, \dots, L_d$  can be computed in  $d$  CONGEST rounds by a simple parallel BFS exploration. Since we assume that  $H$  has no augmenting paths of length shorter than  $d$ , every augmenting path of length  $d$  contains exactly one node from every set  $L_i$  such that the node in  $L_d$  is an unmatched node in  $V_H$ .

We use a variant of the greedy set cover algorithm to find the set  $C$  covering all the shortest augmenting paths in  $H$ . In order to apply the greedy set cover algorithm, we need to know the sizes of the sets  $P_v$ , i.e., for every node  $v$ , we need to know in how many augmenting paths of length  $d$  the node  $v$  is contained. To compute this number, we apply an algorithm that was first developed in [28] and later refined in [5]. The following lemma summarizes the result of [5, 28], for a proof see also the full version of this paper [13].

► **Lemma 7.** [5, 28] *Let  $H = (U_H \cup V_H, E_H)$  be a bipartite graph of maximum degree at most  $\Delta$  and  $M_H$  be a matching of  $H$ . There is a deterministic  $O(d^2)$ -round CONGEST algorithm to compute the number of shortest augmenting paths of length  $d$  passing through every node  $v \in U_H \cup V_H$ .*

We can now use this path counting method to find a small set  $S$  of nodes that covers all augmenting paths of length  $d$ . We start with an empty set  $C$ . The algorithm then works in  $O(d \log \Delta)$  phases  $i = 1, 2, 3, \dots$ , where in phase  $i$ , we add unmatched nodes  $v$  and matching edges  $e$  to  $C$  such that are still contained in at least  $\Delta^d / 2^i$  remaining paths. In order to obtain a polylogarithmic running time, we need to add nodes and edges to  $C$  in parallel. In order to make sure that we do not cover the same path twice, when adding nodes and edges in parallel, we essentially iterate through the  $d$  levels in each phase. The details of the algorithm are given in the following.

**Covering Paths of Length  $d$ : Phase  $i \geq 1$** 

Iterate over all odd levels  $\ell = 1, 3, \dots, d$ :

1. Count the number of augmenting paths of length  $d$  passing through each of the remaining nodes and edges.
2. If  $\ell \in \{1, d\}$ , for all remaining nodes  $v \in L_\ell$  that are in  $p_v \geq \Delta^d/2^i$  different augmenting paths of length  $d$ , add  $v$  to  $C$  and remove  $v$  and its incident edges from  $G_H$  for the remainder of the algorithm.
3. If  $\ell \in \{2, \dots, d-1\}$ , for all remaining matching edges  $e \in M_H$  connecting two nodes  $u \in L_{\ell-1}$  and  $v \in L_\ell$  that are in  $p_e \geq \Delta^d/2^i$  different augmenting paths of length  $d$ , add  $e$  to  $C$  and remove  $e$  and its incident edges from  $G_H$  for the remainder of the algorithm.

Define  $S_H$  to contain every node in  $C$  and both nodes of every edge in  $C$ .

► **Lemma 8.** *Let  $\delta \in (0, 1)$  and assume that  $M_H$  is a  $(1 - \delta)$ -approximate matching of the bipartite graph  $H$  of maximum degree at most  $\Delta$ . Then, the set  $S_H$  selected by the above algorithm has size at most  $\alpha_d \delta \cdot \text{OPT}_H$ , where  $\alpha_d = 2(d+3)(1 + d \ln \Delta)$  and  $\text{OPT}_H$  is the size of a maximum matching and thus of a minimum vertex cover of  $H$ . The time complexity of the algorithm in the CONGEST model is  $O(d^4 \log \Delta)$ .*

**Proof.** We first look at the time complexity of the algorithm in the CONGEST model. The algorithm consists of  $O(d \log \Delta)$  phases, in each phase, we iterate over  $O(d)$  levels and in each of these iterations, the most expensive step is to count the number of augmenting paths passing through each node and edge. By Lemma 7, this can be done in time  $O(d^2)$ , resulting in an overall time complexity of  $O(d^4 \log \Delta)$ .

For each free node  $v \in U_H \cup V_H$  and for each matching edge  $e \in M_H$ , let  $p_v$  and  $p_e$  be the number of (uncovered) augmenting paths of length  $d$  passing through  $v$  and  $e$ , respectively. We will next show that our algorithm is simulating a version of the standard sequential greedy set cover algorithm. When applying the sequential greedy algorithm, in each step, we would need to choose a set  $P_v$  or  $P_e$  of paths that maximizes the number of uncovered augmenting paths of length  $d$  the set covers. We will see that we essentially relax the greedy step and we obtain an algorithm that is equivalent to a sequential algorithm that always picks a set of paths that contains at least half as many uncovered paths as possible. To show this, we first show that for each phase  $i$ , at the beginning of the phase, we have  $p_v, p_e \leq \Delta^d/2^{i-1}$  for all unmatched nodes  $v$  and matching edges  $e$ . For the sake of contradiction, assume that this is not the case and let  $i'$  be the first phase, in which it is not true. Because every node and edge can be contained in at most  $\Delta^d$  augmenting paths of length  $d$ , the statement is definitely true for the first phase and we therefore have  $i' > 1$ . We now consider phase  $i' - 1$ . In each phase, by iterating over all odd levels  $\ell = 1, 3, \dots, d$ , we iterate over all unmatched nodes  $v \in U_H \cup V_H$  and all matching edges  $e \in M_H$  that are contained in some augmenting path of length  $d$ . For each of them, we add the corresponding set  $P_v$  or  $P_e$  to the set cover if we still have  $p_v \geq \Delta/2^{i'-1}$  or  $p_e \geq \Delta/2^{i'-1}$ . At the end of phase  $i' - 1$ , we therefore definitely have  $p_v, p_e < \Delta/2^{i'-1}$  for all nodes  $v$  and matching edges  $e$ , which contradicts the assumption that at the beginning of phase  $i'$ , it is not true that  $p_v, p_e \leq \Delta/2^{i'-1}$  for all such  $v$  and  $e$ . Because in each phase  $i$ , we only add set  $P_v$  and  $P_e$  that are contained in at least  $\Delta/2^i$  uncovered paths, we clearly always pick sets that cover at least half as many uncovered paths as the best current set. Note also that because we iterate through the levels and only add sets for nodes or edges on the same level in parallel, the set that we add in parallel cover disjoint sets of paths. The algorithm is therefore equivalent to a sequential algorithm that adds the sets in each parallel step in an arbitrary order.

Now, we will show that we remove at most  $2(d+3)(1+d\ln\Delta)\delta \cdot \text{OPT}_H$  nodes from graph  $H$ . Indeed, approximating the set cover problem using the standard greedy algorithm gives a  $(1+\ln(s))$  approximation to the solution, where  $s$  is the cardinality of the largest set. If we relax the greedy step by at least a factor of two, as our algorithm does, a standard analysis implies that we still get a  $2(1+\ln s)$ -approximation of the corresponding minimum set cover problem, where  $s$  is still defined as the cardinality of the largest set. In our case, the largest set  $P_v$  or  $P_e$  is  $s \leq \Delta^d$ . Now if the solution to the set cover problem using this greedy version algorithm is  $S_H$  and the optimal solution of the set cover problem is  $S^*$ , then  $|S^*| \leq |S_H| \leq 2(1+d\ln\Delta)|S^*|$ . Recall that  $P_e$  corresponds to a matched edge and by step 3 in our algorithm, both of these matched nodes are removed from the graph  $H$ . Hence, we remove up to  $2|S_H| \leq 4(1+d\ln\Delta)|S^*|$  nodes from  $H$ .

Next, we give an upper bound to  $|S^*|$ , which will finish up our proof. Recall that a solution to our set cover problem is a set of matched edges  $S_e$  and a set of unmatched nodes  $S_v$  that cover all augmenting paths of length  $d$  in  $H$ , i.e., all paths in  $\mathcal{P}$ . Luckily, there is a simple solution to the given set cover problem that allows us to upper bound  $|S^*|$ . We just select a maximal set  $P$  of vertex-disjoint augmenting paths of length  $d$  and we consider all the unmatched nodes and matched edges on these paths to be our solution  $S'$ , where  $|S'| = \frac{d+3}{2}|P|$ . Clearly,  $S'$  is a set cover (and thus  $|S^*| \leq |S'|$ ), as otherwise there would be an augmenting path of length  $d$  that is not covered by  $S'$ . This path has to be vertex-disjoint from all the paths in  $P$ , which is a contradiction to the assumption that  $P$  is a maximal set of vertex-disjoint augmenting paths of length  $d$ . Let  $|M_H^*|$  denote the maximum cardinality of a matching of graph  $H$ . Now, since  $M_H$  is a  $(1-\delta)$ -approximate matching, we can clearly have at most  $\delta|M_H^*|$  vertex-disjoint augmenting paths of at most length  $d$ . Hence, the size of  $P$  can never exceed  $\delta|M_H^*|$  i.e.  $|P| \leq \delta|M_H^*|$ . Thus,  $|S^*| \leq |S'| \leq \frac{d+3}{2}\delta|M_H^*|$ . Hence, we remove at most  $2|S_H| \leq 4(1+d\ln\Delta)|S'| \leq 4(1+d\ln\Delta)\frac{d+3}{2}\delta|M_H^*| \leq 2(d+3)(1+d\ln\Delta)\delta|M_H^*| = 2(d+3)(1+d\ln\Delta)\delta \cdot \text{OPT}_H$  nodes from graph  $H$ . ◀

By iterating over the lengths of shortest paths, we now directly get the following lemma. For a formal proof of the lemma, we refer to the full version of this paper [13].

► **Lemma 9.** *Let  $G = (U \cup V, E)$  be a bipartite graph, let  $k \geq 1$  be an integer parameter, and assume that  $M$  is a  $(1-\delta)$ -approximate matching of  $G$  for some  $\delta \in [0, 1]$ . Further, let  $\text{OPT}$  be the size of a minimum vertex cover of  $G$ . If the bipartition of the nodes of  $G$  into  $U$  and  $V$  is given, there is an  $O(k^5 \log \Delta)$ -time algorithm to compute a node set  $S_1 \subseteq U \cup V$  of size at most  $4k(k+1)(1+2k\ln\Delta)\delta \cdot \text{OPT}$  such that in the induced subgraph  $G[U \cup V \setminus S_1]$ , there is no augmenting path of length at most  $2k-1$  w.r.t. the matching  $\bar{M}$ , where  $\bar{M} \subseteq M$  consists of the edges of  $M$  that connect two nodes in  $U \cup V \setminus S_1$ .*

We now have everything that we need to also get a deterministic  $O(D + \text{poly}(\frac{\log n}{\varepsilon}))$ -time CONGEST algorithm for computing a  $(1+\varepsilon)$ -approximate solution for the MVC problem in bipartite graphs.

► **Theorem 10.** *Let  $G = (V, E)$  be a bipartite graph of diameter  $D$  and maximum degree  $\Delta$  and let  $G' = (V', E')$  be a subgraph of  $G$ . For  $\varepsilon \in (0, 1]$ , there is a deterministic algorithm that gives a  $(1+\varepsilon)$ -approximate minimum vertex cover of graph  $G'$  in  $O(D + \frac{\log^4 n}{\varepsilon^8})$  rounds in the CONGEST model on  $G$ .*

**Proof.** As a first step, we choose a sufficiently small parameter  $\delta > 0$  and we compute a  $(1-\delta)$ -approximate solution  $M'$  to the maximum matching problem on  $G'$  by using the deterministic CONGEST algorithm of [1]. For computing such a matching, the algorithm of [1] has a time complexity of  $O(\frac{\log^2 \Delta + \log^* n}{\delta} + \frac{\log \Delta}{\delta^2}) = O(\frac{\log^2 n}{\delta^2})$ . Let  $k' := \lceil 2/\varepsilon \rceil$  as discussed above. By

## 29:12 Approximating Bipartite Minimum Vertex Cover in the CONGEST Model

Lemma 9, there is a value  $\alpha = 4k'(k' + 1)(1 + 2k' \ln \Delta) = O(k'^3 \log \Delta)$  such that we can find a set  $S_1 \subseteq V'$  of size  $|S_1| = \alpha \delta \text{OPT}$ , where  $\text{OPT}$  is the size of a minimum vertex cover of  $G'$ , such that the following is true. The set  $S_1$  can be computed in time  $O(k'^5 \log \Delta) = O(\frac{\log n}{\varepsilon^5})$ . Let  $G'' = G'[V' \setminus S_1]$  be the induced subgraph of  $G'$  after removing all the nodes in  $S_1$  and let  $M''$  be the subset of the edges in  $M'$  that connect two nodes in  $V' \setminus S_1$  (i.e.,  $M''$  is a matching of  $G''$ ). Then, the graph  $G''$  has no augmenting paths of length at most  $2k' - 1$ . By using Lemma 5, we can therefore compute a  $(1 + 1/k')$ -approximate vertex cover  $S_2$  (and thus a  $(1 + \varepsilon/2)$ -approximate vertex cover) of  $G''$  in time  $O(D + k') = O(D + 1/\varepsilon)$ . Because a minimum vertex cover of  $G''$  is clearly not larger than a minimum vertex cover of  $G'$ , we therefore have  $|S_2| \leq (1 + \varepsilon/2) \cdot \text{OPT}$ . Note that  $S_1 \cup S_2$  is a vertex cover of  $G'$ . The size of  $S_1 \cup S_2$  can be bounded as  $|S_1 \cup S_2| \leq \delta \alpha \cdot \text{OPT} + (1 + \varepsilon/2) \cdot \text{OPT}$ . In order to make sure that this is at most  $(1 + \varepsilon) \cdot \text{OPT}$ , we have to choose  $\delta \leq \varepsilon/(2\alpha)$ . The time complexity to compute the initial matching  $M'$  of  $G'$  is therefore  $O(\frac{\log^2 n}{\delta^2}) = O(\frac{\log^4 n}{\varepsilon^8})$ . ◀

### 4 Polylogarithmic-Time Algorithms

We next show how we can use the algorithms of the previous section together with existing low-diameter graph clustering techniques to obtain polylogarithmic-time approximation schemes for the minimum vertex cover algorithm in the CONGEST model. First we describe a general framework for achieving a  $(1 + \varepsilon)$ -approximate minimum vertex cover  $C$  of unweighted bipartite graphs via an efficient algorithm in the CONGEST model based on a given clustering with some specific properties (cf. Section 2 for the corresponding definitions). We will do so by proving the following lemma. Note that our general framework applies to both the randomized and the deterministic case.

► **Lemma 11.** *Let  $G = (V, E)$  be a bipartite graph and assume that we are given a maximal matching  $M$  of  $G$ . We define edge weights  $w(e) \in \{0, 1\}$  such that  $w(e) = 1$  if and only if  $e \in M$ . Further, assume that w.r.t. those edge weights, we are given a  $(1 - \eta)$  dense, 3-hop separated, and  $(c, d)$ -routable clustering of  $G$ , for some  $\eta \in (0, 1]$  and some positive integers  $c, d > 0$ . Then, for any  $\psi \in (0, 1]$ , we can find a  $(1 + 2\eta + \psi)$ -approximate minimum vertex cover by a deterministic CONGEST algorithm in  $O(c \cdot (d + \text{poly}(\frac{\log n}{\psi})))$  rounds and by a randomized CONGEST algorithm in  $O(c \cdot (d + \frac{\log n}{\psi^3}))$  rounds, w.h.p.*

**Proof.** Let  $\{S_1, S_2, \dots, S_t\}$  be the collection of clusters of the given 3-hop separated,  $(1 - \eta)$ -dense clustering. Define  $E'$  to be the set of edges for which both endpoints are located outside clusters and let  $E''$  to be the set of edges where exactly one of the endpoints is outside clusters. We also say that  $e$  is an edge outside clusters if it is in  $E' \cup E''$ . Further, let  $X$  to be the set of all matched nodes (w.r.t. the given maximal matching  $M$ ) that are outside clusters. Note that since  $M$  is a maximal matching, any edge in  $E'$  is necessarily incident to at least one matched node of  $M$ . Therefore, when adding the set  $X$  to the vertex cover  $C$ , we cover all edges in  $E'$  and possibly some extra edges in  $E''$ . Now since  $G$  is  $(1 - \eta)$ -dense, then at most  $\eta|M|$  matched edges are outside clusters, and when assuming that  $|M^*|$  is the size of a maximum matching of  $G$ , we can deduce that  $|X| \leq 2\eta|M| \leq 2\eta|M^*| = 2\eta\text{OPT}$ , where  $\text{OPT}$  is the size of a minimum vertex cover of  $G$ . Next, we extend each cluster  $S_i$  by at most one hop in radius as follows. For every edge  $\{u, v\} \in E''$  such that  $u \in S_i$  and  $v \notin S_i$ , we add the edge  $\{u, v\}$  and node  $v$  to the cluster. Let  $\{S'_1, S'_2, \dots, S'_t\}$  be the new collection of extended clusters. All edges of  $G$  that are not already covered by  $X$  are now

inside some cluster. In addition, we grow the height of each cluster tree  $T_i$  by at most one hop so that they include the new cluster nodes. We denote the new extended trees by  $T'_i$ . Note that clearly, each edge in  $E$  is still in at most  $c$  trees. Hence, the new collection of extended clusters are now 2-hop separated and  $(c, d + 1)$ -routable.

For each cluster  $S'_i$ , let  $G'_i$  be the graph consisting of the nodes and edges of the cluster. We note that because the clusters are 1-hop separated, the graphs  $G'_i$  are vertex and edge disjoint. In addition, for each cluster  $S'_i$ , we define the graph  $G_i$  as the union of  $G'_i$  and the tree  $T'_i$ . Because the clustering is  $(c, d + 1)$ -routable, it follows that every edge of  $G$  is used by at most  $c$  of the graph  $G_i$  and that the diameter of each graph  $G_i$  is at most  $d + 1$ . To obtain a vertex cover of all edges of  $G$ , we now compute a  $(1 + \psi)$ -approximate minimum vertex cover  $C_i$  for each extended cluster graph  $G'_i$  by running the algorithms described in Theorems 6 and 10. We do this for all clusters in parallel. For each cluster  $S'_i$ , we use  $G_i$  and  $G'_i$  as the graphs  $G$  and  $G'$  in Theorems 6 and 10. Because each edge is contained in at most  $c$  graphs  $G_i$ , we can in parallel run  $T$ -round algorithms in all graphs  $G_i$  in time  $c \cdot T$ . The time complexities therefore follow directly as claimed from the respective time complexities in Theorems 6 and 10.

We define  $Y := \bigcup_{i=1}^t C_i$ . Because every edge of  $G$  that is not covered by the nodes in  $X$  is inside one of the clusters  $S'_i$ , clearly, the set  $X \cup Y$  is a vertex cover of  $G$ . We already showed that  $|X| \leq 2\eta \text{OPT}$ . To bound the size of  $X \cup Y$ , it remains to bound the size of  $Y$ . Let  $\text{OPT}_i$  be the size of an optimal vertex cover of  $G'_i$ . Because the cluster graphs  $G'_i$  are vertex-disjoint, all edges in  $G'_i$  clearly have to be covered by some node of the cluster  $S'_i$  and thus edges in different clusters have to be covered by disjoint sets of nodes. If  $\text{OPT}$  is the size of an optimal vertex cover of  $G$ , we thus clearly have  $\bigcup_{i=1}^t \text{OPT}_i \leq \text{OPT}$ . Because  $C_i$  is a  $(1 + \psi)$ -approximate vertex cover of  $G'_i$ , we also have  $|C_i| \leq (1 + \psi) \cdot \text{OPT}_i$ . Together, we therefore directly get that  $|Y| \leq (1 + \psi) \cdot \text{OPT}$  and therefore  $|X \cup Y| \leq (1 + 2\eta + \psi) \cdot \text{OPT}$ . ◀

In order to prove our two main results, Theorems 2 and 3, we will next show how to efficiently compute the clusterings that are required for Lemma 11. Both clusterings can be obtained by minor adaptations of existing clustering techniques.

## 4.1 The Randomized Clustering

We start with describing the randomized clustering algorithm. By using the exponentially shifted shortest paths approach of Miller, Peng, and Xu [30], we obtain the following lemma.

► **Lemma 12.** *Let  $G = (V, E, w)$  be a weighted bipartite graph with non-negative edge weights  $w(e)$ . For  $\lambda \in (0, 1]$ , there is a randomized algorithm that computes a 3-hop separated clustering of  $G$  such that w.h.p., the clustering is  $(1, O(\frac{\log n}{\lambda}))$ -routable and can be computed in  $O(\frac{\log n}{\lambda})$  rounds in the CONGEST model and such that the clustering is  $(1 - \lambda)$ -dense in expectation.*

The proof of Lemma 12 is a relatively simple adaptation of the clustering algorithm of [30]. For a proof, see the full version of this paper [13].

We now have everything that we need to prove our first main result, our randomized polylogarithmic-time approximation scheme for the MVC problem in bipartite graphs.

**Proof of Theorem 2.** Let  $G = (V, E)$  be the given bipartite graph for which we want to approximate the MVC problem. We first compute a maximal matching  $M$  of  $G$ , which we can for example do by using Luby's algorithm [2, 29] in  $O(\log n)$  rounds. By using  $M$ , we then apply Lemma 12 with  $\lambda = \varepsilon/4$  to obtain a 3-hop separated  $(1, O(\frac{\log n}{\varepsilon}))$ -routable



clustering that is  $(1 - \varepsilon/4)$ -dense in expectation. The time for computing the clustering is  $O\left(\frac{\log n}{\varepsilon}\right)$ , w.h.p. By applying Lemma 11 with  $\eta = \varepsilon/4$  and  $\psi = \varepsilon/2$ , we then get a vertex cover of  $G$  in  $O\left(\frac{\log n}{\varepsilon^3}\right)$  CONGEST rounds such that the expected size of the vertex cover is at most  $(1 + \varepsilon) \cdot \text{OPT}$ , where OPT is the size of a minimum vertex cover of  $G$ . This concludes the proof of the theorem.  $\blacktriangleleft$

## 4.2 The Deterministic Clustering

We obtain the deterministic version of the necessary clustering by adapting the construction of a single color class of the recent efficient deterministic network decomposition algorithm of Rozhoň and Ghaffari [32].

► **Lemma 13.** *Let  $G = (V, E, w)$  be a weighted bipartite graph with non-negative edge weights  $w(e) \in \{0, 1\}$ . For  $\lambda \in (0, 1]$ , there is a deterministic algorithm that computes an  $(1 - \lambda)$ -dense, 3-hop separated, and  $(O(\log n), O\left(\frac{\log^3 n}{\lambda}\right))$ -routable clustering of  $G$  in  $\text{poly}\left(\frac{\log n}{\lambda}\right)$  rounds in the CONGEST model.*

**Proof.** We assume that  $W := \sum_{w \in E} w(e)$  is the total weight of all edges in  $G$ . Let  $\lambda \in (0, 1]$ . We adapt the weak diameter network decomposition algorithm of Rozhoň and Ghaffari [32] applied to the graph  $G^2$  in the CONGEST model. When applied to  $G^2$ , Theorem 2.12 of [32] shows that the algorithm of [32] computes a decomposition of the nodes  $V$  into clusters of  $O(\log n)$  colors such that any two nodes in different clusters of the same color are at distance at least 3 from each other (in  $G$ ). Each cluster is spanned by a Steiner tree of diameter  $O(\log^3 n)$  such that each edge of  $G$  is used by at most  $O(\log n)$  different Steiner trees for each of the  $O(\log n)$  color classes. For our purpose, we only need to construct the first color class of this decomposition. For the first color class, the proof of Theorem 2.12 of [32] implies that the clusters of the first color are 3-hop separated and that they contain a constant fraction of all the nodes. We need to adapt the construction of the first color class of the algorithm of [32] in two ways. In the following, we only sketch these changes.

First, we adapt the algorithm so that it can handle weights. In the following, we define node weight  $\nu(v) \geq 0$  as follows. For each node  $v$ , we define  $\nu(v)$  as the sum of the weights  $w(e)$  of the edges  $e$  that are incident to  $v$ . Note that this implies that the total weight of all the nodes is  $2W$  and that the total weight of all the nodes that are not clustered is an upper bound on the total weight of all the edges outside clusters (i.e., all the edges, where at most one endpoint is inside a cluster). In the algorithm of [32], the clustering is computed in different steps. In each step, some nodes request to join a different cluster and a cluster accepts these requests if the total number of nodes requesting to join the cluster is large enough compared to the total number of nodes already inside the cluster. If a cluster does not accept the requests, the requesting nodes are deactivated and will not be clustered. The threshold on the number of requests required to accept the requests is chosen such that in the end the weak diameter of the clusters is not too large and at the same time, only a constant fraction of all nodes are deactivated and thus not clustered. In our case, we do not care how many nodes are clustered and unclustered, but we care about the total weight of nodes that are clustered and unclustered. The analysis of [32] however directly also works if we instead compare the total weight of the nodes that request to join a cluster with the total weight of the nodes that are already inside the cluster. If the node weights are polynomially bounded non-negative integers (which they are in our case), the asymptotic guarantees of the construction are exactly the same. In this way, we can make sure to construct  $(O(\log n), O(\log^3 n))$ -routable, 3-hop separated clusters such that a constant fraction of the total weight of all the nodes is inside clusters.



As a second change, in order to make sure that the clustering is also  $(1 - \lambda)$ -dense, we need to guarantee that the total weight of the nodes that are unclustered is at most a  $\lambda/2$ -fraction of the total weight of all the nodes. We can guarantee this, by adapting the threshold for accepting nodes to a cluster. We essentially have to multiply the threshold by a factor  $\Theta(\lambda)$  to make sure that this is the case. This increases the maximal possible cluster diameter by a factor  $O(1/\lambda)$  and it increases the total running time by a factor  $\text{poly}(1/\lambda)$ . ◀

*Remark:* In the above lemma, we assumed for simplicity that the edge weights are either 0 or 1. The construction however directly also works in the same way and with the same asymptotic guarantees if the edge weights are polynomially bounded non-negative integers. With some simple preprocessing, one can also obtain the same asymptotic result for arbitrary non-negative edge weights.

In a similar way as we proved Theorem 2, we can now also prove our second main result, our deterministic polylogarithmic-time approximation scheme for the MVC problem in bipartite graphs.

**Proof of Theorem 3.** Let  $G = (V, E)$  be the given bipartite graph for which we want to approximate the MVC problem. We first compute a maximal matching  $M$  of  $G$ , which we can do by using the algorithm of Fischer [15] in  $O(\log^2 \Delta \cdot \log n)$  deterministic rounds in the CONGEST model. By using  $M$ , we then apply Lemma 13 with  $\lambda = \varepsilon/4$  to obtain a  $(1 - \varepsilon/4)$ -dense, 3-hop separated  $(O(\log n), \text{poly}(\frac{\log n}{\varepsilon}))$ -routable clustering. By Lemma 13, the time for computing the clustering in the CONGEST model is  $\text{poly}(\frac{\log n}{\varepsilon})$ . By applying Lemma 11 with  $\eta = \varepsilon/4$  and  $\psi = \varepsilon/2$ , we then get a  $(1 + \varepsilon)$ -approximate vertex cover of  $G$  in  $\text{poly}(\frac{\log n}{\varepsilon})$  CONGEST rounds, which completes the proof of the theorem. ◀

---

## References

- 1 M. Ahmadi, F. Kuhn, and R. Oshman. Distributed approximate maximum matching in the CONGEST model. In *Proc. 32nd Symp. on Distributed Computing (DISC)*, pages 6:1–6:17, 2018.
- 2 N. Alon, L. Babai, and A. Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *Journal of Algorithms*, 7(4):567–583, 1986.
- 3 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.
- 4 N. Bachrach, K. Censor-Hillel, M. Dory, Y. Efron, D. Leitersdorf, and A. Paz. Hardness of distributed optimization. In *Proc. 38th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 238–247, 2019.
- 5 R. Bar-Yehuda, K. Censor-Hillel, M. Ghaffari, and G. Schwartzman. Distributed approximation of maximum independent set and maximum matching. *CoRR*, abs/1708.00276, 2017. Conference version at PODC 2017.
- 6 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.
- 7 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 *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 3–8, 2016.
- 8 L. Barenboim, M. Elkin, S. Pettie, and J. Schneider. The locality of distributed symmetry breaking. In *Proceedings of 53th Symposium on Foundations of Computer Science (FOCS)*, 2012.

- 9 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.
- 10 G. E. Blelloch, A. Gupta, I. Koutis, G. L. Miller, R. Peng, and K. Tangwongsan. Nearly-linear work parallel SDD solvers, low-diameter decomposition, and low-stretch subgraphs. *Theory Comput. Syst.*, 55(3):521–554, 2014.
- 11 K. Censor-Hillel, S. Houry, and A. Paz. Quadratic and near-quadratic lower bounds for the CONGEST model. In *Proc. 31st Symp. on Distributed Computing (DISC)*, pages 10:1–10:16, 2017.
- 12 R. Diestel. *Graph Theory*, chapter 2.1, pages 35–58. Springer, Berlin, 3rd edition, 2005.
- 13 S. Faour and F. Kuhn. Approximate bipartite vertex cover in the CONGEST model. *CoRR*, abs/2011.10014, 2020.
- 14 U. Feige, Y. Mansour, and R. E. Schapire. Learning and inference in the presence of corrupted inputs. In *Proc. 28th Conf. on Learning Theory (COLT)*, pages 637–657, 2015.
- 15 M. Fischer. Improved deterministic distributed matching via rounding. In *Proc. 31st Symp. on Distributed Computing (DISC)*, pages 17:1–17:15, 2017.
- 16 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.
- 17 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.
- 18 M. Göös and J. Suomela. No sublogarithmic-time approximation scheme for bipartite vertex cover. *Distributed Computing*, 27(6):435–443, 2014.
- 19 F. Grandoni, J. Könemann, and A. Panconesi. Distributed weighted vertex cover via maximal matchings. *ACM Trans. Algorithms*, 5(1):6:1–6:12, 2008.
- 20 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.
- 21 J. E. Hopcroft and R. M. Karp. An  $n^{5/2}$  algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 1973.
- 22 A. Israeli and A. Itai. A fast and simple randomized parallel algorithm for maximal matching. *Inf. Process. Lett.*, 22(2):77–80, 1986.
- 23 D. König. Gráfok és mátrixok. *Matematikai és Fizikai Lapok*, 38:116–119, 1931.
- 24 Subhash Khot and Oded Regev. Vertex cover might be hard to approximate to within 2-epsilon. *J. Comput. Syst. Sci.*, 74(3):335–349, 2008.
- 25 F. Kuhn, T. Moscibroda, and R. Wattenhofer. What cannot be computed locally! In *Proceedings of 23rd ACM Symposium on Principles of Distributed Computing (PODC)*, pages 300–309, 2004.
- 26 F. Kuhn, T. Moscibroda, and R. Wattenhofer. The price of being near-sighted. In *Proceedings of 17th Symposium on Discrete Algorithms (SODA)*, pages 980–989, 2006.
- 27 N. Linial and M. Saks. Low diameter graph decompositions. *Combinatorica*, 13(4):441–454, 1993.
- 28 Z. Lotker, B. Patt-Shamir, and S. Pettie. Improved distributed approximate matching. *J. ACM*, 62(5):38:1–38:17, 2015.
- 29 M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 15:1036–1053, 1986.
- 30 G. L. Miller, R. Peng, and S. C. Xu. Parallel graph decompositions using random shifts. In *Proc. 25th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 196–203, 2013.
- 31 D. Peleg. *Distributed Computing: A Locality-Sensitive Approach*. SIAM, 2000.
- 32 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.

# Distributed Distance Approximation

**Bertie Ancona**

MIT, Cambridge, MA, USA  
bancona@alum.mit.edu

**Keren Censor-Hillel**

Technion, Haifa, Israel  
ckeren@cs.technion.ac.il

**Mina Dalirrooyfard**

MIT, Cambridge, MA, USA  
minad@mit.edu

**Yuval Efron**

Technion, Haifa, Israel  
efronyuv@gmail.com

**Virginia Vassilevska Williams**

MIT, Cambridge, MA, USA  
virgi@mit.edu

---

## Abstract

Diameter, radius and eccentricities are fundamental graph parameters, which are extensively studied in various computational settings. Typically, computing approximate answers can be much more efficient compared with computing exact solutions. In this paper, we give a near complete characterization of the trade-offs between approximation ratios and round complexity of distributed algorithms for approximating these parameters, with a focus on the weighted and directed variants.

Furthermore, we study *bi-chromatic* variants of these parameters defined on a graph whose vertices are colored either red or blue, and one focuses only on distances for pairs of vertices that are colored differently. Motivated by applications in computational geometry, bi-chromatic diameter, radius and eccentricities have been recently studied in the sequential setting [Backurs et al. STOC'18, Dalirrooyfard et al. ICALP'19]. We provide the first distributed upper and lower bounds for such problems.

Our technical contributions include introducing the notion of *approximate pseudo-center*, which extends the *pseudo-centers* of [Choudhary and Gold SODA'20], and presenting an efficient distributed algorithm for computing approximate pseudo-centers. On the lower bound side, our constructions introduce the usage of new functions into the framework of reductions from 2-party communication complexity to distributed algorithms.

**2012 ACM Subject Classification** Theory of computation

**Keywords and phrases** Distributed Computing, Distance Computation, Algorithms, Lower Bounds

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2020.30

**Related Version** A full version of this paper is available at <https://arxiv.org/abs/2011.05066>.

**Funding** This project has received funding from the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme under grant agreement No 755839.

## 1 Introduction

The diameter and radius are central graph parameters, defined as the maximum and minimum eccentricities over all vertices, respectively, where the eccentricity of a vertex  $v$  is the maximum distance out of  $v$ . Computing the diameter and radius of a given graph are cornerstone



© Bertie Ancona, Keren Censor-Hillel, Mina Dalirrooyfard, Yuval Efron, and Virginia Vassilevska Williams;

licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 30; pp. 30:1–30:17



Leibniz International Proceedings in Informatics

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

problems with abundant applications. This is particularly the case in the context of distributed computing, where distances between nodes in a network (and in particular the graph diameter) directly influence the time it takes to communicate throughout the network.

We focus on computing the diameter, radius and eccentricities in the classic CONGEST model of distributed computation, in which  $n$  nodes of a synchronous network communicate by exchanging messages of  $O(\log n)$  bits with their neighbors in the underlying network graph. In a seminal work, Frischknecht et al. [35] showed that the diameter is hard to compute in CONGEST, namely that  $\tilde{\Omega}(n)^1$  rounds are required, even in undirected unweighted graphs. Abboud et al. [1] showed that the same holds for computing the radius. Both of these results are tight up to logarithmic factors due to algorithms that compute all pairs shortest paths (APSP) in a given unweighted, undirected graph in  $O(n)$  rounds, see Lenzen and Peleg, and Peleg et al. [50, 53]. Recently, Bernstein and Nanongkai [14], presented an algorithm which computes exact APSP in a given weighted, directed graph in  $\tilde{O}(n)$  rounds as well.

As computing the diameter and radius exactly in general graphs is hard, a natural relaxation is to settle for approximate computations. In an unweighted, undirected graph, a simple observation due to the triangle inequality is that computing a BFS tree from any node yields a 2-approximation to the diameter or radius, and a 3-approximation of all eccentricities.

Obtaining a more thorough understanding of the complexity landscape of computing approximations to these distance parameters has been an ongoing endeavour of the community. The current state of the art for diameter approximation is the algorithm by Holzer et al. [40] with round complexity of  $O(\sqrt{n \log n} + D)$ , that achieves a  $\frac{3}{2}$ -approximation of the diameter in a given unweighted, undirected graph (further discussion is deferred to Section 1.2).

However, many open cases have remained, and unveiling the full picture of the trade-offs between approximation ratios and round complexity for distance parameters in the CONGEST model has remained a central open problem. In this paper, we give a near-complete characterization of this trade-off for the problems of diameter, radius and eccentricities, focusing on the weighted and/or directed variants. For the problem of directed diameter, only the range  $[\frac{3}{2}, 2]$  of approximation ratios remains open.

In some cases, originally motivated by computational geometry problems [4, 29, 46, 57], we are interested in a “bi-chromatic” definition of the parameters. In the bi-chromatic setting, the vertices are partitioned into two sets,  $S$  and  $T = V \setminus S$ , and the bi-chromatic eccentricity of a node  $s \in S$  is the maximum distance from  $s$  to a node in  $T$ . The bi-chromatic diameter and radius are the maximum and minimum bi-chromatic eccentricities of nodes in  $S$ .

The bi-chromatic versions of diameter and radius have received much recent attention in the sequential setting [11, 24]. In this paper, we initiate the study of these problems in the CONGEST model, by providing upper and lower bounds for these problems. For example, we prove that a  $\frac{5}{3}$ -approximation to bi-chromatic diameter in an unweighted, undirected graph can be computed in  $\tilde{O}(\sqrt{n} + D)$  rounds, and we prove this is tight in the sense that any improvement in the approximation ratio incurs a blowup in the round complexity to  $\tilde{\Omega}(n)$ .

A more comprehensive display of our results follows. Also, a comparison with previous work is depicted in Table 1 and Table 2 and is elaborated upon in Section 1.2.

## 1.1 Our contributions and techniques

As mentioned earlier, the *eccentricity*  $ecc(v)$  of a vertex  $v$  is the distance  $\max_{u \in V} d(v, u)$ . The *diameter*  $D$  is the largest eccentricity in the graph, and the *radius*  $r$  is the smallest.

---

<sup>1</sup> Throughout the paper,  $\tilde{O}$  and  $\tilde{\Omega}$  are used to hide poly-logarithmic factors

■ **Table 1** Upper bounds for the problems considered in this paper. A variant can be weighted, directed, both, or neither. Upper bounds hold for the listed variants and all subsets of those variants. Approximation factors are multiplicative but may omit additive error. The value  $k$  may be any integer greater or equal to 1. We denote the round complexity of the current best exact weighted SSSP algorithm by  $T(SSSP)$ , currently  $\tilde{O}(\min\{\sqrt{nD}, \sqrt{nD}^{\frac{1}{4}} + n^{\frac{3}{5}}\} + D)$  by [34].

\*for  $k = 1$

Problem	Approx.	Variant	Upper Bound $\tilde{O}(\cdot)$	Reference
Diameter	Exact	wted dir	$n$	[14]
	$2 - \frac{1}{2^k}$		$n^{\frac{1}{k+1}} + D$	Theorem 9, [39]*
	2	wted dir	$T(SSSP)$	Corollary 4
	$2 + \epsilon$	wted	$\sqrt{n} + D$	[13]
wted dir		$\sqrt{nD}^{1/4} + D$	Corollary 3	
Radius	Exact	wted dir	$n$	[14]
	$2 - \frac{1}{2^k}$		$n^{\frac{1}{k+1}} + D$	Theorem 9
	2	wted dir	$T(SSSP)$	Corollary 4
	$2 + \epsilon$	wted	$\sqrt{n} + D$	Corollary 2
wted dir		$\sqrt{nD}^{1/4} + D$	Corollary 3	
Eccentricities	Exact	wted dir	$n$	[14]
	$3 - \frac{4}{2^k+1}$		$n^{\frac{1}{k+1}} + D$	Theorem 9
	2	wted dir	$T(SSSP)$	Corollary 4
	$2 + \epsilon$	wted	$\sqrt{n} + D$	Corollary 2
wted dir		$\sqrt{nD}^{1/4} + D$	Corollary 3	
Bi-chromatic Diameter	Exact	wted dir	$n$	[14]
	$5/3$		$\sqrt{n} + D$	Theorem 10
	2	wted	$T(SSSP)$	Theorem 11

**Directed/weighted Radius and Eccentricities.** We present a connection between the complexity of computing or approximating the Single Source Shortest Paths (SSSP) problem and the complexity of approximating radius, diameter and eccentricities. Formally, we prove the following theorem in Section 3.

► **Theorem 1.** *For any  $\epsilon \geq 0$ , given a  $(1 + \epsilon)$ -approximation algorithm  $\mathcal{A}_\epsilon$  for weighted and directed SSSP running in  $T(n, \epsilon, D)$  rounds, there exists an algorithm for  $(2 + \epsilon^3 + 3\epsilon^2 + 4\epsilon)$ -approximate diameter, radius, and all eccentricities in  $\tilde{O}(T(n, \epsilon, D) + D)$  rounds on weighted, directed graphs.*

We now describe the challenges in proving the above and how we cope with them. A useful notion for distance parameters is the *center* of a graph, which is the vertex with the lowest eccentricity. Given the center  $c$  of a graph, we can easily approximate all eccentricities of a given graph by performing an SSSP algorithm rooted at  $c$ , and letting each node  $v$  estimate its eccentricity by outputting  $d(v, c) + ecc(c)$ . However, computing the center of a graph, or even its eccentricity (the radius), is a hard task that requires  $\tilde{\Omega}(n)$  rounds [1].

For proving Theorem 1, we rely on an approach of Choudhary and Gold [22]. Here, one defines a notion of a *pseudo-center* and one then shows how to compute a pseudo-center of size  $O(\log^2 n)$  sequentially in near-linear time. A pseudo-center  $C$  is a set of nodes, whose goal is to mimic the center of the graph, by promising that all eccentricities are at least the maximal distance between any node to the pseudo-center  $C$ . Using such a pseudo-center, one estimates the eccentricity of every node, similarly to the case of computing the actual center.

■ **Table 2** Lower bounds for the problems considered in this paper. A variant can be weighted, directed, both, or neither. Lower bounds hold for the listed variants and all supersets of those variants. Approximation factors are multiplicative.

Problem	Approx.	Variant	Lower Bound $\tilde{\Omega}(\cdot)$	Reference
Diameter	$3/2 - \varepsilon$		$n$	[1]
	$2 - \varepsilon$	wted		[41]
	poly( $n$ )	wted	$\sqrt{n} + D$	[49]
		dir		Theorem 8
Radius	$3/2 - \varepsilon$		$n$	[1]
	$2 - \varepsilon$	wted		Theorem 5
		dir		
	poly( $n$ )	wted	$\sqrt{n} + D$	Corollary 6
		dir		
Eccentricities	$5/3 - \varepsilon$		$n$	[1]
	$2 - \varepsilon$	wted		[41]
		dir		Theorem 5
	poly( $n$ )	wted	$\sqrt{n} + D$	Corollary 6
		dir		
Bi-chromatic Diameter	$5/3 - \varepsilon$		$n$	Theorem 14
	$2 - \varepsilon$	wted		[41]
		dir		Theorem 15
	poly( $n$ )	wted	$\sqrt{n} + D$	Corollary 6
		dir		

The algorithm of [22] for computing a small pseudo-center can be viewed as a reduction to Single Source Shortest Paths (SSSP), which is very efficient in the sequential setting. However, the current state-of-the-art *distributed* complexity of computing exact SSSP is very costly, and hence we wish to avoid it. To overcome this, we introduce the notion of an *approximate pseudo-center*, which generalizes the notion of a pseudo-center. We prove that (i) an approximate pseudo-center of small size can be computed efficiently in a distributed manner (thus avoiding the complexities of exact SSSP), and (ii) an approximate pseudo-center is still sufficient for approximating the required distance parameters.

From Theorem 1, using the  $(1 + \varepsilon)$ -approximate SSSP algorithms of [13, 34], which run in  $\tilde{O}((\sqrt{n} + D)/\varepsilon)$  rounds on weighted, undirected graphs and  $\tilde{O}((\sqrt{n}D^{1/4} + D)/\varepsilon)$  rounds on weighted, directed graphs, respectively, we deduce the following corollaries:

► **Corollary 2.** *For any  $\varepsilon = 1/\text{polylog}(n)$ , there exists an algorithm for  $(2 + \varepsilon)$ -approximate diameter, radius and all eccentricities running in  $\tilde{O}(\sqrt{n} + D)$  rounds on nonnegative weighted graphs, with  $n$  nodes and hop-diameter  $D$ .*

► **Corollary 3.** *For any  $\varepsilon = 1/\text{polylog}(n)$ , there exists an algorithm for  $(2 + \varepsilon)$ -approximate diameter, radius and all eccentricities running in  $\tilde{O}(\sqrt{n}D^{1/4} + D)$  rounds on nonnegative weighted, directed graphs, with  $n$  nodes and hop-diameter  $D$ .*

Using the exact SSSP algorithm of Chechik and Mukhtar [21] we obtain the following.

► **Corollary 4.** *There exists an algorithm for 2-approximate radius, diameter and all eccentricities running in  $\tilde{O}(\sqrt{n}D^{1/4} + D)$  rounds on nonnegative weighted, directed graphs, with  $n$  nodes and hop-diameter  $D$ .*

Regarding radius, the only previous result regarding the complexity of approximating the radius in the CONGEST model is due to [1], in which they showed that for any  $\varepsilon > 0$ , computing an  $(3/2 - \varepsilon)$ -approximation to the radius in undirected, unweighted graphs requires



$\tilde{\Omega}(n)$  rounds. Abboud et al. [1] show that any algorithm computing an  $(\frac{5}{3} - \varepsilon)$ -approximation of all eccentricities requires  $\tilde{\Omega}(n)$  rounds as well. Having a complete understanding of the relationship between approximation ratio and the round complexity of computing unweighted, undirected radius remains an intriguing open problem. As a step towards resolving this problem, we give a nearly full characterization of the approximation factor to round complexity mapping for radius in *weighted* or *directed* graphs in the CONGEST model.

In Section 4 we prove the following.

► **Theorem 5.** *Given any constant  $\varepsilon > 0$ , any algorithm (even randomized) computing an  $(2 - \varepsilon)$ -approximation to the weighted (directed) radius in a given weighted (directed) graph  $G$  requires  $\tilde{\Omega}(n)$  rounds.*

A standard technique for proving lower bounds for the CONGEST model, is to reduce it from 2-party communication complexity. In the context of the distance parameters discussed in this work, this framework was used by [35] to show that any algorithm that distinguishes between networks with diameter 2 and 3 requires  $\tilde{\Omega}(n)$  rounds. Later, [1] showed that this lower bound holds even when one considers sparse networks with only  $O(n)$  edges (they also proved more results as discussed in the related work section).

Many of the papers that employ this framework, reduce from either the Set Disjointness function, the Equality function, or the Gap Disjointness function [10, 16, 23, 25]. In this work, we enhance this framework by showing lower bounds using reductions from other functions, which were not used previously to obtain lower bounds for the CONGEST model. Namely, in the proof of Theorem 5, we use the Tribes function, defined by Jayram et al. in [45], and the Hitting Set Existence (HSE) function, which is a communication complexity variant of a problem introduced by Abboud et al. in [3]. We elaborate upon this framework and the functions that we use in Section 2.

The following is a corollary of Theorem 7 and Theorem 8 which are stated below for the diameter, since any finite approximation to the radius, implies a finite approximation to the diameter, as  $r \leq D \leq 2r$ .

► **Corollary 6.** *Given any positive function  $\alpha(n)$ , any algorithm (even randomized) computing an  $\alpha(n)$ -approximation to the weighted (directed) radius in a given weighted (directed) graph  $G$  requires  $\tilde{\Omega}(\sqrt{n} + D)$  rounds.*

**Directed/Weighted Diameter.** In previous work, Holzer and Pinski [41] showed a lower bound of  $\tilde{\Omega}(n)$  rounds for computing a  $(2 - \varepsilon)$ -approximation of the diameter of a given weighted graph. Shortly after, Becker et al. [13] designed an algorithm that computes a  $(2 + o(1))$ -approximation of weighted and directed diameter in  $\tilde{O}(\sqrt{n}D^{1/4} + D)$  rounds. Such an algorithm makes one wonder, is there a smooth trade-off between the round complexity and the approximation ratio when going beyond a 2-approximation, for either the directed or weighted variants? In other words, can one further reduce the round complexity if we are willing to settle for a worse approximation ratio? For weighted diameter, this question was resolved by Lenzen et al. [49] in the negative, in the sense that the dependence on  $n$  in the algorithm of [13] is necessary (up to poly-logarithmic factors) for any approximation of the diameter in weighted or directed graphs. We give a proof of this result for completeness, and this allows us to more easily present a similar new result for the *bi-chromatic* diameter case. The bi-chromatic diameter is a variant of the diameter problem that is discussed later.

► **Theorem 7.** *Given any positive function  $\alpha(n)$ , any algorithm (even randomized) computing an  $\alpha(n)$ -approximation to the weighted diameter or bi-chromatic diameter in a given graph  $G$  requires  $\tilde{\Omega}(\sqrt{n} + D)$  rounds.*



► **Theorem 8.** *Given any positive function  $\alpha(n)$ , any algorithm (even randomized) computing an  $\alpha(n)$ -approximation to the diameter in a given directed graph  $G$  requires  $\tilde{\Omega}(\sqrt{n} + D)$  rounds.*

To prove these theorems we reduce from the problem of Spanning Connected Subgraph Verification (SCSV) to approximating these parameters. The SCSV problem is known to admit the above lower bound due to Das Sarma et al. [25]. The key challenge is to construct a reduction in a manner that can be efficiently simulated in CONGEST. The proofs of these theorems are given in full version of the paper.

**Undirected and Unweighted Diameter, Radius and Eccentricities.** Abboud et al. [1] show that for any  $\varepsilon > 0$ , any algorithm computing an  $(\frac{3}{2} - \varepsilon)$ -approximation of diameter or radius in unweighted undirected graphs has round complexity  $\tilde{\Omega}(n)$ . Furthermore, any algorithm computing an  $(\frac{5}{3} - \varepsilon)$ -approximation to all eccentricities has round complexity  $\tilde{\Omega}(n)$ . For upper bounds, the state of art for diameter approximation is an algorithm by Holzer et al. [40], computing a  $3/2$ -approximation in  $\tilde{O}(\sqrt{n \log n} + D)$  rounds. Fully understanding the mapping of approximation ratios in the range  $[\frac{3}{2}, 2)$  for diameter and radius, and in the range  $(\frac{5}{3}, 3)$  for all eccentricities, to their respective correct round complexity in the CONGEST model remains open. As a step towards resolving this open problem, in the full version of the paper, we present a simple distributed implementation of a sequential approximation algorithm of Cairo et al. [15] for diameter, radius and eccentricities with the following parameters.

► **Theorem 9.** *For any  $k \in \mathbb{N}$ , there exist algorithms that compute  $(2 - \frac{1}{2^k})$ -approximate diameter and radius and  $(3 - \frac{4}{2^k+1})$ -approximate eccentricities on unweighted, undirected graphs, that have running time of  $\tilde{O}(n^{\frac{1}{k+1}} + D)$  rounds w.h.p.*

**Bi-chromatic Diameter and Radius.** To the best of our knowledge, no previous results regarding bi-chromatic distance parameters are known in distributed settings. Roughly speaking, these variants are defined using only distances between pairs of nodes in  $S \times T$  where  $S, T \subseteq V, T = V \setminus S$ .  $D_{ST}, R_{ST}$  respectively denote the  $ST$ -diameter  $\max_{s \in S, t \in T} d(s, t)$  and the  $ST$ -radius  $\min_{s \in S} \max_{t \in T} d(s, t)$  (also see Section 2.1). In the following,  $T(SSSP)$  refers to the distributed complexity of exact weighted SSSP. The proofs of these theorems can be found in the full version of the paper

► **Theorem 10.** *There is an algorithm with complexity  $\tilde{O}(\sqrt{n} + D)$  that given an undirected, unweighted graph  $G = (V, E)$ , and sets  $S \subseteq V, T = V \setminus S$ , w.h.p. computes a value  $D_{ST}^*$  such that  $\frac{3D_{ST}}{5} - \frac{6}{5} \leq D_{ST}^* \leq D_{ST}$ .*

► **Theorem 11.** *There is an algorithm with complexity  $T(SSSP)$  that given an undirected graph  $G = (V, E)$ , and sets  $S \subseteq V, T = V \setminus S$ , computes a value  $D^*$  such that  $\frac{D_{ST}}{2} - W/2 \leq D^* \leq D_{ST}$ . Here  $W$  is the minimum edge weight in  $S \times T$ .*

We remark that using very similar algorithms to the ones of Theorem 10 and Theorem 11, one can obtain the following results, whose proofs we omit due to similarity to the main ideas in the proofs we provide for the above two theorems.

► **Remark 12.** There are algorithms with complexity  $\tilde{O}(\sqrt{n} + D)$  that given an undirected, unweighted graph  $G = (V, E)$ , and sets  $S, T \subseteq V$ , compute w.h.p. the following.

1. A value  $R_{ST}^*$  such that  $R_{ST} \leq R_{ST}^* \leq \frac{5R_{ST}}{3} + \frac{5}{3}$ , in the case that  $S = V \setminus T$ .
2. A 2-approximation to all  $ST$ -eccentricities.
3. A 2-approximation to  $R_{ST}$ .

► **Remark 13.** There are algorithms with complexity  $T(SSSP)$  that given an undirected graph  $G = (V, E)$ , and sets  $S, T \subseteq V$ , compute the following.

1. A value  $R_{ST}^*$  such that  $R_{ST} \leq R_{ST}^* \leq 2R_{ST} + W$ , in the case that  $S = V \setminus T$ . Here  $W$  is the minimum edge weight in  $S \times T$ .
2. A 3-approximation to all  $ST$ -eccentricities.
3. A 3-approximation to  $R_{ST}$ .

We complement these upper bounds with several lower bounds. We show that in the weighted case, one cannot hope to do better than a  $\frac{5}{3}$ -approximation for bi-chromatic diameter with  $O(n^{1-\epsilon})$  rounds for some  $\epsilon > 0$ . Additionally, as a step towards realizing the complexity of finding a better than 2-approximation for directed diameter, we show that for *bi-chromatic* diameter, in which one is tasked with finding the largest distance between a pair of nodes in different sets of a given partition of the graph, finding such an approximation is a hard task. Formally, we prove the following theorems in the full version of the paper due to lack of space.

► **Theorem 14.** *For all constant  $\epsilon > 0$ , there is no  $o(\frac{n}{\log^3 n})$  round algorithm for computing a  $(\frac{5}{3} - \epsilon)$ -approximation to the bi-chromatic diameter in an unweighted, undirected graph.*

► **Theorem 15.** *For all constant  $\epsilon > 0$ , there is no  $o(\frac{n}{\log^2 n})$  round algorithm for computing a  $(2 - \epsilon)$ -approximation to the bi-chromatic diameter in a directed graph.*

Finally, we show that for both the directed and weighted cases, any approximation of the bi-chromatic diameter requires  $\tilde{\Omega}(\sqrt{n} + D)$  rounds. The weighted case is proved as part of Theorem 7. In the full version of the paper we prove separately the directed case, which is stated formally as follows.

► **Theorem 16.** *Given any positive function  $\alpha(n)$ , any algorithm (even randomized) computing an  $\alpha(n)$ -approximation to the bi-chromatic diameter in a given directed graph  $G$  requires  $\tilde{\Omega}(\sqrt{n} + D)$  rounds.*

## 1.2 Additional related work

The state of the art algorithm for 3/2-approximation of unweighted, undirected diameter [40] was preceded by a significant number of works. Notable examples are Holzer's and Wattenhofer's algorithm computing a 3/2-approximation of the diameter in undirected, unweighted graphs in  $O(n^{3/4} + D)$  rounds [42], and the independent work of Peleg et al. [53], which achieves the same approximation in  $O(D\sqrt{n} \log n)$  rounds. Later, Lenzen and Peleg [50] improved this upper bound to  $O(\sqrt{n} \log n + D)$ .

Approximations to more concrete variants of distance computations such as APSP and SSSP have been extensively studied in the CONGEST as well. Examples include the deterministic  $(1 + o(1))$ -approximation to APSP by Nanongkai [52], and the  $(1 + \epsilon)$ -approximation algorithm for SSSP of Becker et al. [13]. The near optimal algorithm of Bernstein and Nanongkai for APSP [14] was preceded by a series of papers that set to realize the complexity of APSP in CONGEST [5, 6, 8, 30, 43]. Given that [14] is a randomized Las Vegas algorithm, there remains a gap between the best known deterministic and randomized algorithms for APSP, with the deterministic state of the art being  $\tilde{O}(n^{4/3})$  [7]. For SSSP, the state of the art algorithm of [21] was also preceded by a series of improvements [13, 30, 31, 34, 37, 38, 48, 52] from the folklore  $O(n)$  Bellman-Ford algorithm.

Approximations to distance computations have been studied in various distributed settings, such as the congested clique model. Starting from [18], which presented the first non trivial algorithms for both exact, and approximated APSP in the model. From there

a series of works designed more and more efficient algorithms for approximating distances in the model [13, 17, 20, 26, 31, 32, 36], with the most recent work being the  $\text{poly}(\log \log n)$  approximations for APSP and Multi Source Shortest Paths [27].

Conditional hardness results for these parameters are very well-studied in the sequential setting, within fine-grained complexity, under assumptions such as the Strong Exponential Time Hypothesis (SETH) [44]. For details, see e.g., the work of Backurs et al. [11] or the survey by Vassilevska Williams [56]. Returning to the CONGEST model, in some topologies such as planar graphs, work by Li and Parter [51] showed that the diameter of an unweighted, undirected graph can even be computed in a sublinear number of rounds.

The lower bound framework for reducing 2-party communication complexity to CONGEST was introduced by Peleg and Rubinfeld in [54], in which they show that any algorithm solving the minimum spanning tree (MST) problem has round complexity  $\tilde{\Omega}(\sqrt{n} + D)$ . Since then, there has been a surge of lower bounds for the CONGEST model employing this framework; examples include [2, 10, 23, 25, 28, 33]. In an independent concurrent work, [9] show another angle of the landscape of the complexity of diameter approximation, proving that for any constant  $\epsilon > 0$ , any algorithm approximating the diameter of a given unweighted, undirected graph, within a factor of  $(\frac{3}{5} + \epsilon)$ ,  $(\frac{4}{7} + \epsilon)$ , or  $(\frac{6}{11} + \epsilon)$ , must have a round complexity of at least  $\tilde{\Omega}(n^{1/3})$ ,  $\tilde{\Omega}(n^{1/4})$ , or  $\tilde{\Omega}(n^{1/6})$ , respectively.

## 2 Preliminaries

### 2.1 The Model & Definitions

This paper considers the CONGEST model of computation. In this model, a synchronized network of  $n$  nodes is represented by an undirected, unweighted, simple graph  $G = (V, E)$ . In each round, each node can send a different message of  $O(\log n)$  bits to each of its neighbors.

Next, we define the network parameters that we discuss in the paper.

► **Definition 17.** *Given a weighted, directed graph  $G = (V, E)$ , denote by  $d(u, v)$  the weight of the lightest directed path starting at node  $u$  and ending at node  $v$ . If there is no such path, we define  $d(u, v) = \infty$ . Here, the weight of a path  $P$  is the sum of the weights of its edges. The eccentricity  $\text{ecc}(u)$  of a node  $u$  is defined to be  $\max_{v \in V} d(u, v)$ . The radius  $r$  of  $G$  is defined to be  $\min_{v \in V} \text{ecc}(v)$ . The diameter  $D$  of  $G$  is defined to be  $\max_{v \in V} \text{ecc}(v)$ .*

The  $ST$  variants of these distance parameters are defined as follows.

► **Definition 18** (*ST and bi-chromatic diameter, radius and eccentricities*). *Given a weighted graph  $G = (V, E)$ , and two non empty subsets  $S, T \subseteq V$ , given  $v \in S$ , we define its  $ST$ -eccentricity by  $\text{ecc}(v) = \max_{u \in T} d(v, u)$ . We define the  $ST$ -diameter of  $G$  to be  $D_{ST} = \max_{v \in S} \text{ecc}(v)$ . The  $ST$ -radius of  $G$  is defined to be  $R_{ST} = \min_{v \in S} \text{ecc}(v)$ . When  $S = V \setminus T$ , the  $ST$  parameters are called bi-chromatic.*

### 2.2 The Communication Complexity Framework

The high level idea of applying the framework of reductions from 2-party communication complexity to obtain lower bounds in the CONGEST model is as follows. We pick some function  $f : \{0, 1\}^k \times \{0, 1\}^k \rightarrow \{0, 1\}$ , and then reduce any efficient communication protocol for it to an efficient CONGEST algorithm for the discussed problem. We start with our two players Alice (A) and Bob (B), each of them respectively receives a binary string of length  $k$  denoted by  $x, y \in \{0, 1\}^k$ .

We construct a graph  $G = (V, E)$  we call the *fixed graph construction*, and we partition the set of vertices  $V$  into the sets  $V_A, V_B$ . We call the cut induced by  $V_A, V_B$  the *communication cut*, and we denote the number of edges in this cut by  $|cut|$ .

Now, given  $x$  and the graph  $G[V_A]$  (i.e., the subgraph of  $G$  induced by  $V_A$ ), Alice modifies the graph  $G[V_A]$  in any way that may depend only on  $x$ , and Bob does the same with  $y$  and  $G[V_B]$ . Denote the resulting graph by  $G_{x,y}$ , and denote its number of nodes by  $n$ .

The resulting graph  $G_{x,y}$  should be constructed such that it has some property  $P$  (e.g. radius at least 3) iff  $f(x,y) = 1$ . Now, assuming there is an algorithm  $Alg$  in the CONGEST model that decides  $P$  in  $T$  rounds, Alice and Bob can simulate this algorithm on  $G_{x,y}$ , and the only communication required between them is for simulating messages that are sent on edges in the communication cut. Thus, Alice and Bob can simulate  $Alg(G_{x,y})$  while communicating  $O(T \cdot |cut| \cdot \log n)$  bits of communication. Furthermore, by the property of  $G_{x,y}$ , deciding  $P$  on  $G_{x,y}$  allows them to compute  $f(x,y)$  with  $O(T \cdot |cut| \cdot \log n)$  bits of communication. Therefore, a lower bound on the communication complexity of  $f$ , implies a lower bound on  $T$ , which is the round complexity of the distributed algorithm.

We next elaborate on the functions  $f$  that we use in our reductions.

► **Definition 19** (The Set Disjointness Problem (Disj) [55]). *Alice and Bob receive subsets  $X, Y \subseteq [n]$ , respectively, represented as binary vectors of length  $n$ . Their goal is to decide whether  $X \cap Y = \emptyset$ .*

It is known by [12, 47, 55] that the randomized communication complexity of Disj on inputs of size  $n$  is  $\Omega(n)$ .

► **Definition 20** (The Tribes (ListDISJ) Problem [45]). *Alice and Bob are given sets  $A_i, B_i \in \{0, 1\}^N$  for each  $i \in [N]$ . They must output 1 if and only if there is some  $i$  such that  $A_i$  and  $B_i$  are disjoint, i.e. there is no  $j$  such that  $A_{ij} = B_{ij} = 1$ . We treat the inputs  $x$  and  $y$  as binary strings of length  $N^2$ , such that  $x = A_1 \circ \dots \circ A_N, y = B_1 \circ \dots \circ B_N$ . Here,  $\circ$  refers to string concatenation.*

The Tribes function is defined in [45], where a lower bound of  $\Omega(N^2)$  communication bits is proved, even for randomized protocols.

The full version of the paper contains discussion of additional functions which are employed to prove the results not present in this version.

## 3 Approximation Algorithms

### 3.1 Approximations for weighted directed variants

In this section, we prove our approximation algorithms, starting with the connection between the complexity of *SSSP* and approximating distance parameters. Formally, we prove the following theorem, and then we deduce Corollaries 2, 3, and 4.

**Theorem 1** *For any  $\varepsilon \geq 0$ , given a  $(1 + \varepsilon)$ -approximation algorithm  $\mathcal{A}_\varepsilon$  for weighted and directed *SSSP* running in  $T(n, \varepsilon, D)$  rounds, there exists an algorithm for  $(2 + \varepsilon^3 + 3\varepsilon^2 + 4\varepsilon)$ -approximate diameter, radius, and all eccentricities in  $\tilde{O}(T(n, \varepsilon, D) + D)$  rounds on weighted, directed graphs.*

We briefly remind the reader of the discussion in the introduction regarding the theorem. In order to obtain fast algorithms and maintaining the quality of the approximation, we generalize the notion of *pseudo-center* defined by Choudhary and Gold [22] into *approximate pseudo-center*. We show how to compute such a set of small size, and we show that such a set suffices to obtain the approximations detailed in Theorem 1.

## 30:10 Distributed Distance Approximation

► **Definition 21.** A  $\alpha$ -approximate pseudo-center is a set  $C$  of nodes such that for all nodes  $v \in V$ ,  $\text{ecc}(v) \geq \max_{u \in V} \min_{c \in C} \{d(c, u)/\alpha\}$ .

We begin by showing that we can compute a small approximate pseudo-center efficiently.

► **Lemma 22.** Given a  $(1 + \varepsilon)$ -approximate,  $T(n, \varepsilon, D)$ -round SSSP algorithm  $\mathcal{A}_\varepsilon$ , there is a Las Vegas algorithm to compute a  $(1 + \varepsilon)^2$ -approximate pseudo-center of size  $O(\log^2(n))$  of a graph  $G = (V, E)$  in  $\tilde{O}(T(n, \varepsilon, D))$  rounds of communication, with high probability.

**Proof.** Let the set  $C$  begin empty, and let  $W$  begin as the set  $V$ . Throughout the proof, running  $\mathcal{A}_\varepsilon$  outward (inward) from a vertex  $v \in V$  means computing the distances from  $v$  to the rest of the nodes (to  $v$  from the rest of the nodes). We repeat the following until  $W$  is empty:

- Assign each node in  $W$  to a set  $S$  independently with probability  $\min\{1, 24 \log(n)/|W|\}$ . Resample if  $|S| < 8 \log n$  or  $|S| > 36 \log n$ .
- Run  $\mathcal{A}_\varepsilon$  outward from each node in  $S$ , and for all  $u \in V$ , compute estimated distances  $d_{\mathcal{A}_\varepsilon}(S, u) = \min_{s \in S} \{d_{\mathcal{A}_\varepsilon}(s, u)\}$ .
- Let  $a$  be the node with the largest estimated distance from  $S$ . Then, we broadcast  $d_{\mathcal{A}_\varepsilon}(S, a)$  to all nodes in the graph using some BFS tree.
- Run  $\mathcal{A}_\varepsilon$  inward from  $a$ , and remove all nodes  $u$  where  $d_{\mathcal{A}_\varepsilon}(u, a) \geq d_{\mathcal{A}_\varepsilon}(S, a)$  from  $W$ .
- Add  $S$  to  $C$ .

First, we argue that  $C$  is a  $(1 + \varepsilon)^2$ -approximate pseudo-center. We only remove a node  $u$  from  $W$  when  $d_{\mathcal{A}_\varepsilon}(u, a) \geq d_{\mathcal{A}_\varepsilon}(S, a)$  for some sample  $S$ . Let  $a^*$  be the node that is truly farthest from  $S$ ; then  $d_{\mathcal{A}_\varepsilon}(S, a) \geq d(S, a^*)/(1 + \varepsilon) \geq \max_{x \in V} \min_{c \in C} \{d(c, x)/(1 + \varepsilon)\}$ , because  $S \subseteq C$ . We also note that by similarly bounding the error of  $\mathcal{A}_\varepsilon$ , it holds that  $d_{\mathcal{A}_\varepsilon}(u, a) \leq (1 + \varepsilon)d(u, a) \leq (1 + \varepsilon)\text{ecc}(u)$ , so we may conclude that

$$(1 + \varepsilon)\text{ecc}(u) \geq \max_{x \in V} \min_{c \in C} \{d(c, x)/(1 + \varepsilon)\}.$$

In other words,  $\text{ecc}(u) \geq \max_{x \in V} \min_{c \in C} \{d(c, x)/(1 + \varepsilon)^2\}$ , which meets the definition of a  $(1 + \varepsilon)^2$ -pseudo-center.

Next, we argue that each iteration requires  $\tilde{O}(T(n, \varepsilon, D))$  rounds. Using a Chernoff bound, it is simple to show that in each round,  $8 \log n \leq |S| \leq 36 \log n$  with probability at least  $1 - 1/n^4$ , so we expect to resample a sub-constant number of times. We then run  $\mathcal{A}_\varepsilon$  from each node in  $S$  and we run it again once to the node  $a$ , for a total of  $O(\log n \cdot T(n, \varepsilon, D))$  rounds. The rest of each iteration involves a constant number of broadcasts that take  $O(D)$  rounds in total.

Finally, we argue that with high probability, we only have  $O(\log n)$  iterations in our algorithm. We do this by showing that in iteration  $i$ , the size of  $W$  reduces by at least half with high probability, i.e.  $|W_i|/2 \geq |W_{i+1}|$ . Consider the set  $X \subseteq W_i$  of  $|W_i|/2$  nodes with the smallest  $d_{\mathcal{A}_\varepsilon}(u, a)$ ,  $u \in W_i$ . Note that  $S_i$  is a randomly sampled subset of  $W_i$  of size at least  $8 \log n$ , and thus intersects  $X$  with probability at least  $(1 - 1/n^5)$ , as argued in Lemma 23 below [22] with no further assumptions.

All nodes in  $W_i \setminus X$  are at least as far as any node in that intersection under  $\mathcal{A}_\varepsilon$ , by definition. This implies that for all  $u \in W_i \setminus X$ ,  $d_{\mathcal{A}_\varepsilon}(u, a) \geq d_{\mathcal{A}_\varepsilon}(S, a)$ , which implies that all  $|W_i|/2$  nodes of  $W_i \setminus X$  will be removed from  $W_i$  in iteration  $i$ . ◀

► **Lemma 23** (Lemma 2.1 in [22]). Let  $U$  be a universe set of size at most  $n$ , and let  $S_1, \dots, S_n \subseteq U$  such that  $|S_i| \geq L$  for each  $i \in [n]$ . Let  $c$  be some constant and  $r = \frac{n(c+1) \ln n}{L}$ . Let  $S \subseteq U$  be a random subset of size  $r$ , then it holds that  $S \cap S_i \neq \emptyset$  for all  $i$  with probability  $1 - n^{-c}$ .

Now that we showed how to compute an approximate pseudo-center, we show that it is sufficient for approximating the distance parameters as claimed.

► **Lemma 24.** *Given a  $(1 + \varepsilon)^2$ -approximate pseudo-center  $C$  and a  $(1 + \varepsilon)$ -approximate SSSP algorithm  $\mathcal{A}_\varepsilon$  taking  $T(n, \varepsilon, D)$  rounds, we may compute  $(2 + \varepsilon^3 + 3\varepsilon^2 + 4\varepsilon)$ -approximate eccentricities for all nodes in  $O(|C| \cdot T(n, \varepsilon, D) + D)$  rounds.*

**Proof.** First, we run  $\mathcal{A}_\varepsilon$  to and from each node in  $C$ , so that each node  $v \in V$  stores  $d_{\mathcal{A}_\varepsilon}(c, v)$  and  $d_{\mathcal{A}_\varepsilon}(v, c)$  for all  $c \in C$ . Each node  $u$  internally determines  $\min_{c \in C} \{d_{\mathcal{A}_\varepsilon}(c, u)\}$ . Then, using aggregation over a BFS tree, the nodes determine, and then broadcast the value  $D_{\mathcal{A}_\varepsilon}(C) := \max_{u \in V} \min_{c \in C} \{d_{\mathcal{A}_\varepsilon}(c, u)\}$ . Thus, the aggregation takes  $O(D)$  rounds. Each node  $v$  approximates its eccentricity as  $\max_{c \in C} \{d_{\mathcal{A}_\varepsilon}(v, c)\} + D_{\mathcal{A}_\varepsilon}(C)$ .

First, note that this estimate is at least the true eccentricity of  $v$ , as each computed distance represents some path in the graph, and in this distance a path can go from  $v$  to any node in  $C$  and then any node in  $V$ .

We argue that this is a  $(2 + \varepsilon^3 + 3\varepsilon^2 + 4\varepsilon)$ -approximation. The estimated distance  $\max_{c \in C} \{d_{\mathcal{A}_\varepsilon}(v, c)\}$  is at most  $(1 + \varepsilon) \cdot ecc(v)$ , because  $\mathcal{A}_\varepsilon$  overestimates by at most a factor of  $1 + \varepsilon$ . By our definition of  $(1 + \varepsilon)^2$ -approximate pseudo-center,  $D(C) \leq (1 + \varepsilon)^2 ecc(v)$ . Our estimate  $D_{\mathcal{A}_\varepsilon}(C)$  is at most  $(1 + \varepsilon) \cdot D(C)$ , so  $D_{\mathcal{A}_\varepsilon}(C) \leq (1 + \varepsilon)^3 ecc(v)$ . Thus,  $\max_{c \in C} \{d_{\mathcal{A}_\varepsilon}(v, c)\} + D_{\mathcal{A}_\varepsilon}(C) \leq (1 + \varepsilon + (1 + \varepsilon)^3) \cdot ecc(v) = (2 + \varepsilon^3 + 3\varepsilon^2 + 4\varepsilon) \cdot ecc(v)$ .

We compute  $\mathcal{A}_\varepsilon$  twice for each element of  $C$ , and broadcast a constant number of values to all nodes, so the total number of rounds is  $O(|C| \cdot T(n, \varepsilon, D) + D)$ . ◀

**Proof of Theorem 1.** Applying Lemma 22 and Lemma 24, given a  $(1 + \varepsilon)$ -approximate algorithm  $\mathcal{A}_\varepsilon$  for SSSP running in  $T(n, \varepsilon, D)$  rounds, we may compute  $(2 + \varepsilon^3 + 3\varepsilon^2 + 4\varepsilon)$ -approximations for all eccentricities in  $O(\log^2(n) \cdot T(n, \varepsilon, D) + D)$  rounds. ◀

Using the  $(1 + \varepsilon)$ -approximate SSSP algorithms of [13, 34], which run in  $\tilde{O}((\sqrt{n} + D)/\varepsilon)$  rounds on weighted, undirected graphs and  $\tilde{O}((\sqrt{n}D^{1/4} + D)/\varepsilon)$  rounds on weighted, directed graphs respectively, we achieve the following corollaries:

► **Corollary 2.** *For any  $\varepsilon = 1/\text{polylog}(n)$ , there exists an algorithm for  $(2 + \varepsilon)$ -approximate diameter, radius and all eccentricities running in  $\tilde{O}(\sqrt{n} + D)$  rounds on nonnegative weighted graphs, with  $n$  nodes and hop-diameter  $D$ .*

► **Corollary 3.** *For any  $\varepsilon = 1/\text{polylog}(n)$ , there exists an algorithm for  $(2 + \varepsilon)$ -approximate diameter, radius and all eccentricities running in  $\tilde{O}(\sqrt{n}D^{1/4} + D)$  rounds on nonnegative weighted, directed graphs, with  $n$  nodes and hop-diameter  $D$ .*

Using the exact SSSP algorithm of [21], which runs in  $\tilde{O}(\sqrt{n}D^{1/4} + D)$  rounds, we obtain the following corollary.

► **Corollary 4.** *There exists an algorithm for 2-approximate radius, diameter and all eccentricities running in  $\tilde{O}(\sqrt{n}D^{1/4} + D)$  rounds on nonnegative weighted, directed graphs, with  $n$  nodes and hop-diameter  $D$ .*

## 4 Hardness of Approximation

In this section, we prove the lower bound results of the paper. As stated, we use reductions from 2-party communication complexity. To formalize the reductions, we restate the following definition from Censor-Hillel et al. [19].



► **Definition 25** (Family of Lower Bound Graphs). *Given integers  $K$  and  $n$ , a Boolean function  $f : \{0, 1\}^K \times \{0, 1\}^K \rightarrow \{0, 1\}$  and some Boolean graph property or predicate denoted  $P$ , a set of graphs  $\{G_{x,y} = (V, E_{x,y}) \mid x, y \in \{0, 1\}^K\}$  is called a family of lower bound graphs with respect to  $f$  and  $P$  if the following hold:*

1. *The set of vertices  $V$  is the same for all the graphs in the family, and we denote by  $V_A, V_B$  a fixed partition of the vertices.*
2. *Given  $x, y \in \{0, 1\}^K$ , the only part of the graph which is allowed to be dependent on  $x$  (by adding edges or weights, no adding vertices) is  $G[V_A]$ .*
3. *Given  $x, y \in \{0, 1\}^K$ , the only part of the graph which is allowed to be dependent on  $y$  (by adding edges or weights, no adding vertices) is  $G[V_B]$ .*
4.  *$G_{x,y}$  satisfies  $P$  if and only if  $f(x, y) = 1$ .*

*The set of edges  $E(V_A, V_B)$  is denoted by  $E_{cut}$ , and is the same for all graphs in the family.*

We use the following theorem whose proof can be found in Censor-Hillel et al. [19], with  $CC^R(f)$  denoting the randomized communication complexity of  $f$ .

► **Theorem 26.** *Fix a function  $f : \{0, 1\}^K \times \{0, 1\}^K \rightarrow \{0, 1\}$  and a predicate  $P$ . If there exists a family of lower bound graphs  $\{G_{x,y}\}$  w.r.t  $f$  and  $P$ , then every randomized algorithm for deciding  $P$  takes  $\Omega(CC^R(f)/(|E_{cut}| \log n))$  rounds.*

#### 4.1 Lower bounds for radius

We start with proving our two lower bounds for weighted or directed radius approximations.

We divide the proof of Theorem 5 into two cases which we prove separately. We prove the weighted case here, and the proof of the directed case appears in the full version of the paper.

**Theorem 5 [Weighted case]** *For any  $\varepsilon = 1/\text{poly}(n)$ ,  $(2 - \varepsilon)$ -approximation of the radius of a weighted graph with  $n$  nodes requires  $\Omega(n/\log n)$  rounds, even when the graph has constant hop-diameter.*

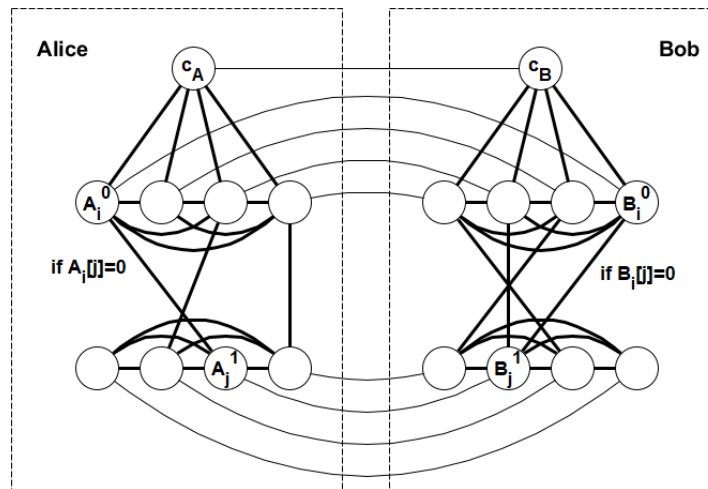
**Proof.** We reduce from the Tribes problem with vector sets  $A$  and  $B$  of size  $N$ . This construction is similar to that of [41, Theorem 7].

Figure 1 illustrates our family of lower bound graphs. We construct four cliques  $A^0, A^1, B^0, B^1$  of size  $N$ , where the edges of the cliques have weight  $t$ , a value we will set later. Let  $K_i$  be the  $i$ th node in clique  $K$ . Add two nodes  $c_A$  and  $c_B$ .

Connect all nodes in  $A^0$  to  $c_A$  with edges of weight  $t$ , and connect all nodes in  $B^0$  to  $c_B$  with edges of weight  $t$ . Connect  $c_A$  and  $c_B$  with an edge of weight 1. For all  $i \in [N]$  and  $b \in \{0, 1\}$ , connect  $A_i^b$  and  $B_i^b$  with an edge of weight 1. Connect  $A_i^0$  and  $A_j^1$  with an edge of weight  $t$  if and only if  $A_i[j] = 0$ . Connect  $B_i^0$  and  $B_j^1$  with an edge of weight  $t$  if and only if  $B_i[j] = 0$ . Alice will simulate the nodes  $A^0 \cup A^1 \cup \{c_A\}$ , and Bob will simulate the nodes  $B^0 \cup B^1 \cup \{c_B\}$ .

First, we claim that if  $(A, B)$  is a “yes” instance of Tribes, then the radius is at most  $t + 2$ . To show this, note that in this case, there must be some  $i$  such that the  $i$ th vectors of  $A$  and  $B$  are orthogonal. Consider the node  $A_i^0$ . It may reach in distance at most  $t + 1$  all nodes in  $B^0 \cup A^0$ , via a clique edge and an edge in the matching between  $A^0$  and  $B^0$ . It may also reach  $\{c_A, c_B\}$  in at most  $t + 1$ . It may also reach all nodes in  $A^1 \cup B^1$  in distance at most  $t + 2$ , because for any  $j$  where  $A_i[j] = 0$  or  $B_i[j] = 0$ , either  $A_i^0$  may reach  $A_j^1$  in distance  $t$  or  $B_i^0$  may reach  $B_j^1$  in distance  $t$ . Since  $A_i$  and  $B_i$  are orthogonal, this is true for all  $j$ . Thus the eccentricity of  $A_i^0$  is at most  $t + 2$ , which upper-bounds the radius.





■ **Figure 1** Sketch of Theorem 5, weighted case construction. Bold lines represent edges of weight  $t$ .

Second, we claim that if  $(A, B)$  is a “no” instance of Tribes, then the radius is at least  $2t$ . To see this, first note that  $c_A$  and  $c_B$  have eccentricity at least  $2t$ , because that is the shortest possible distance between them and  $B^1 \cup A^1$ . By the same argument, the eccentricity of all nodes in  $A^1 \cup B^1$  is also at least  $2t$ . For all  $i$ ,  $A_i$  and  $B_i$  are not orthogonal, which means that for all  $i$  there is some  $j$  such that neither  $A_i^0$  nor  $B_i^0$  has an edge to  $B_j^1$  or  $A_j^1$ . Clearly any other path from  $B_i^0$  or  $A_i^0$  to  $B_j^1$  or  $A_j^1$  is at least of length  $2t$ , via a clique edge of weight  $t$ . Thus the eccentricities of all nodes are at least  $2t$ , so the radius is at least  $2t$ .

We set  $t = \lceil \frac{4}{\varepsilon} \rceil$  so that a  $(2 - \varepsilon)$ -approximate radius algorithm needs to distinguish between  $t + 2$  and  $2t$ . The constructed graph  $G_{A,B}$  has  $n = O(N)$  nodes with a cut of size  $O(n)$ , which by Theorem 26 and the lower bound of  $\Omega(N^2)$  for the communication complexity of Tribes, implies that the radius algorithm requires  $\Omega(n/\log n)$  rounds. ◀

## References

- 1 Amir Abboud, Keren Censor-Hillel, and Seri Khoury. Near-linear lower bounds for distributed distance computations, even in sparse networks. In *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings*, pages 29–42, 2016. doi:10.1007/978-3-662-53426-7\_3.
- 2 Amir Abboud, Keren Censor-Hillel, Seri Khoury, and Ami Paz. Smaller cuts, higher lower bounds. *CoRR*, abs/1901.01630, 2019. arXiv:1901.01630.
- 3 Amir Abboud, Virginia Vassilevska Williams, and Joshua Wang. Approximation and fixed parameter subquadratic algorithms for radius and diameter in sparse graphs. In *Proceedings of the Twenty-seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '16, pages 377–391, Philadelphia, PA, USA, 2016. Society for Industrial and Applied Mathematics. URL: <http://dl.acm.org/citation.cfm?id=2884435.2884463>.
- 4 Pankaj K. Agarwal, Herbert Edelsbrunner, and Otfried Schwarzkopf. Euclidean minimum spanning trees and bichromatic closest pairs. *Discret. Comput. Geom.*, 6:407–422, 1991. doi:10.1007/BF02574698.
- 5 Udit Agarwal and Vijaya Ramachandran. New and simplified distributed algorithms for weighted all pairs shortest paths. *CoRR*, abs/1810.08544, 2018. arXiv:1810.08544.
- 6 Udit Agarwal and Vijaya Ramachandran. Distributed weighted all pairs shortest paths through pipelining. In *2019 IEEE International Parallel and Distributed Processing Symposium*,

- IPDPS 2019, Rio de Janeiro, Brazil, May 20-24, 2019*, pages 23–32. IEEE, 2019. doi:10.1109/IPDPS.2019.00014.
- 7 Udit Agarwal and Vijaya Ramachandran. Faster deterministic all pairs shortest paths in congest model. In Christian Scheideler and Michael Spear, editors, *SPAA '20: 32nd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, July 15-17, 2020*, pages 11–21. ACM, 2020. doi:10.1145/3350755.3400256.
  - 8 Udit Agarwal, Vijaya Ramachandran, Valerie King, and Matteo Pontecorvi. A deterministic distributed algorithm for exact weighted all-pairs shortest paths in  $\tilde{O}(n^{3/2})$  rounds. In Calvin Newport and Idit Keidar, editors, *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom, July 23-27, 2018*, pages 199–205. ACM, 2018. doi:10.1145/3212734.3212773.
  - 9 Anonymous authors. Improved hardness of approximation of diameter in the congest model. In *submission*, 2020.
  - 10 Nir Bachrach, Keren Censor-Hillel, Michal Dory, Yuval Efron, Dean Leitersdorf, and Ami Paz. Hardness of distributed optimization. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 238–247, 2019. doi:10.1145/3293611.3331597.
  - 11 Arturs Backurs, Liam Roditty, Gilad Segal, Virginia Vassilevska Williams, and Nicole Wein. Towards tight approximation bounds for graph diameter and eccentricities. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018*, pages 267–280, New York, NY, USA, 2018. ACM. doi:10.1145/3188745.3188950.
  - 12 Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, and D. Sivakumar. An information statistics approach to data stream and communication complexity. *J. Comput. Syst. Sci.*, 68(4):702–732, 2004. doi:10.1016/j.jcss.2003.11.006.
  - 13 Ruben Becker, Andreas Karrenbauer, Sebastian Krinninger, and Christoph Lenzen. Near-Optimal Approximate Shortest Paths and Transshipment in Distributed and Streaming Models. In Andréa W. Richa, editor, *31st International Symposium on Distributed Computing (DISC 2017)*, volume 91 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.DISC.2017.7.
  - 14 Aaron Bernstein and Danupon Nanongkai. Distributed exact weighted all-pairs shortest paths in near-linear time. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019*, pages 334–342, New York, NY, USA, 2019. ACM. doi:10.1145/3313276.3316326.
  - 15 Massimo Cairo, Roberto Grossi, and Romeo Rizzi. New bounds for approximating extremal distances in undirected graphs. In *Proceedings of the Twenty-seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '16*, pages 363–376, Philadelphia, PA, USA, 2016. Society for Industrial and Applied Mathematics. URL: <http://dl.acm.org/citation.cfm?id=2884435.2884462>.
  - 16 Keren Censor-Hillel and Michal Dory. Distributed spanner approximation. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom, July 23-27, 2018*, pages 139–148, 2018. doi:10.1145/3212734.3212758.
  - 17 Keren Censor-Hillel, Michal Dory, Janne H. Korhonen, and Dean Leitersdorf. Fast approximate shortest paths in the congested clique. In Peter Robinson and Faith Ellen, editors, *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 74–83. ACM, 2019. doi:10.1145/3293611.3331633.
  - 18 Keren Censor-Hillel, Petteri Kaski, Janne H. Korhonen, Christoph Lenzen, Ami Paz, and Jukka Suomela. Algebraic methods in the congested clique. *Distributed Comput.*, 32(6):461–478, 2019. doi:10.1007/s00446-016-0270-2.
  - 19 Keren Censor-Hillel, Seri Khoury, and Ami Paz. Quadratic and near-quadratic lower bounds for the CONGEST model. In *31st International Symposium on Distributed Computing, DISC*

- 2017, October 16-20, 2017, Vienna, Austria, pages 10:1–10:16, 2017. doi:10.4230/LIPIcs.DISC.2017.10.
- 20 Shiri Chechik and Doron Mukhtar. Reachability and shortest paths in the broadcast CONGEST model. In Jukka Suomela, editor, *33rd International Symposium on Distributed Computing, DISC 2019, October 14-18, 2019, Budapest, Hungary*, volume 146 of *LIPIcs*, pages 11:1–11:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.DISC.2019.11.
- 21 Shiri Chechik and Doron Mukhtar. Single-source shortest paths in the CONGEST model with improved bound. In Yuval Emek and Christian Cachin, editors, *PODC '20: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, August 3-7, 2020*, pages 464–473. ACM, 2020. doi:10.1145/3382734.3405729.
- 22 Keerti Choudhary and Omer Gold. Extremal distances in directed graphs: Tight spanners and near-optimal approximation algorithms. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 495–514. SIAM, 2020. doi:10.1137/1.9781611975994.30.
- 23 Artur Czumaj and Christian Konrad. Detecting cliques in CONGEST networks. In *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*, pages 16:1–16:15, 2018. doi:10.4230/LIPIcs.DISC.2018.16.
- 24 Mina Dalirrooyfard, Virginia Vassilevska Williams, Nikhil Vyas, and Nicole Wein. Tight approximation algorithms for bichromatic graph diameter and related problems. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, volume 132 of *LIPIcs*, pages 47:1–47:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ICALP.2019.47.
- 25 Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. Distributed verification and hardness of distributed approximation. In *Proceedings of the Forty-third Annual ACM Symposium on Theory of Computing, STOC '11*, pages 363–372, New York, NY, USA, 2011. ACM. doi:10.1145/1993636.1993686.
- 26 Michael Dinitz and Yasamin Nazari. Massively parallel approximate distance sketches. In Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller, editors, *23rd International Conference on Principles of Distributed Systems, OPODIS 2019, December 17-19, 2019, Neuchâtel, Switzerland*, volume 153 of *LIPIcs*, pages 35:1–35:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.OPODIS.2019.35.
- 27 Michal Dory and Merav Parter. Exponentially faster shortest paths in the congested clique. *CoRR*, abs/2003.03058, 2020. arXiv:2003.03058.
- 28 Andrew Drucker, Fabian Kuhn, and Rotem Oshman. On the power of the congested clique model. In *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*, pages 367–376, 2014. doi:10.1145/2611462.2611493.
- 29 Adrian Dumitrescu and Sumanta Guha. Extreme distances in multicolored point sets. *J. Graph Algorithms Appl.*, 8:27–38, 2004. doi:10.7155/jgaa.00080.
- 30 Michael Elkin. Distributed exact shortest paths in sublinear time. In Hamed Hatami, Pierre McKenzie, and Valerie King, editors, *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 757–770. ACM, 2017. doi:10.1145/3055399.3055452.
- 31 Michael Elkin and Ofer Neiman. Hopsets with constant hopbound, and applications to approximate shortest paths. *SIAM J. Comput.*, 48(4):1436–1480, 2019. doi:10.1137/18M1166791.
- 32 Michael Elkin and Ofer Neiman. Linear-size hopsets with small hopbound, and constant-hopbound hopsets in RNC. In Christian Scheideler and Petra Berenbrink, editors, *The 31st ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2019, Phoenix, AZ, USA, June 22-24, 2019*, pages 333–341. ACM, 2019. doi:10.1145/3323165.3323177.
- 33 Orr Fischer, Tzvil Gonen, Fabian Kuhn, and Rotem Oshman. Possibilities and impossibilities for distributed subgraph detection. In *Proceedings of the 30th on Symposium on Parallelism in*

- Algorithms and Architectures, SPAA 2018, Vienna, Austria, July 16-18, 2018*, pages 153–162, 2018. doi:10.1145/3210377.3210401.
- 34 Sebastian Forster and Danupon Nanongkai. A faster distributed single-source shortest paths algorithm. In Mikkel Thorup, editor, *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018, Paris, France, October 7-9, 2018*, pages 686–697. IEEE Computer Society, 2018. doi:10.1109/FOCS.2018.00071.
  - 35 Silvio Frischknecht, Stephan Holzer, and Roger Wattenhofer. Networks cannot compute their diameter in sublinear time. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 1150–1162, 2012. doi:10.1137/1.9781611973099.91.
  - 36 François Le Gall. Further algebraic algorithms in the congested clique model and applications to graph-theoretic problems. In Cyril Gavoille and David Ilcinkas, editors, *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings*, volume 9888 of *Lecture Notes in Computer Science*, pages 57–70. Springer, 2016. doi:10.1007/978-3-662-53426-7\_5.
  - 37 Mohsen Ghaffari and Jason Li. Improved distributed algorithms for exact shortest paths. In Ilias Diakonikolas, David Kempe, and Monika Henzinger, editors, *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, June 25-29, 2018*, pages 431–444. ACM, 2018. doi:10.1145/3188745.3188948.
  - 38 Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. A deterministic almost-tight distributed algorithm for approximating single-source shortest paths. In Daniel Wichs and Yishay Mansour, editors, *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 489–498. ACM, 2016. doi:10.1145/2897518.2897638.
  - 39 Stephan Holzer, David Peleg, Liam Roditty, and Roger Wattenhofer. Brief announcement: Distributed  $3/2$ -approximation of the diameter. In *Distributed Computing - 28th International Symposium, DISC 2014*, pages 562–564, 2014.
  - 40 Stephan Holzer, David Peleg, Liam Roditty, and Roger Wattenhofer. Distributed  $3/2$ -approximation of the diameter. In *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings*, pages 562–564, 2014. URL: <http://link.springer.com/content/pdf/bbm%3A978-3-662-45174-8%2F1.pdf>.
  - 41 Stephan Holzer and Nathan Pinsker. Approximation of Distances and Shortest Paths in the Broadcast Congest Clique. In Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Butucaru, editors, *19th International Conference on Principles of Distributed Systems (OPODIS 2015)*, volume 46 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1–16, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.OPODIS.2015.6.
  - 42 Stephan Holzer and Roger Wattenhofer. Optimal distributed all pairs shortest paths and applications. In *ACM Symposium on Principles of Distributed Computing, PODC '12, Funchal, Madeira, Portugal, July 16-18, 2012*, pages 355–364, 2012. doi:10.1145/2332432.2332504.
  - 43 Chien-Chung Huang, Danupon Nanongkai, and Thatchaphol Saranurak. Distributed exact weighted all-pairs shortest paths in  $\tilde{O}(n^{5/4})$  rounds. In Chris Umans, editor, *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, October 15-17, 2017*, pages 168–179. IEEE Computer Society, 2017. doi:10.1109/FOCS.2017.24.
  - 44 Russell Impagliazzo and Ramamohan Paturi. On the complexity of  $k$ -sat. *J. Comput. Syst. Sci.*, 62(2):367–375, 2001. doi:10.1006/jcss.2000.1727.
  - 45 T. S. Jayram, Ravi Kumar, and D. Sivakumar. Two applications of information complexity. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing, June 9-11, 2003, San Diego, CA, USA*, pages 673–682, 2003. doi:10.1145/780542.780640.
  - 46 Naoki Katoh and Kazuo Iwano. Finding  $k$  farthest pairs and  $k$  closest/farthest bichromatic pairs for points in the plane. *Int. J. Comput. Geometry Appl.*, 5:37–51, 1995. doi:10.1142/S0218195995000040.

- 47 Eyal Kushilevitz and Noam Nisan. *Communication Complexity*. Cambridge University Press, New York, NY, USA, 1997.
- 48 Christoph Lenzen and Boaz Patt-Shamir. Fast routing table construction using small messages: extended abstract. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pages 381–390. ACM, 2013. doi:10.1145/2488608.2488656.
- 49 Christoph Lenzen, Boaz Patt-Shamir, and David Peleg. Distributed distance computation and routing with small messages. *Distributed Comput.*, 32(2):133–157, 2019. doi:10.1007/s00446-018-0326-6.
- 50 Christoph Lenzen and David Peleg. Efficient distributed source detection with limited bandwidth. In *ACM Symposium on Principles of Distributed Computing, PODC '13, Montreal, QC, Canada, July 22-24, 2013*, pages 375–382, 2013. doi:10.1145/2484239.2484262.
- 51 Jason Li and Merav Parter. Planar diameter via metric compression. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, pages 152–163, 2019. doi:10.1145/3313276.3316358.
- 52 Danupon Nanongkai. Distributed approximation algorithms for weighted shortest paths. In David B. Shmoys, editor, *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 565–573. ACM, 2014. doi:10.1145/2591796.2591850.
- 53 David Peleg, Liam Roditty, and Elad Tal. Distributed algorithms for network diameter and girth. In *Automata, Languages, and Programming - 39th International Colloquium, ICALP 2012, Warwick, UK, July 9-13, 2012, Proceedings, Part II*, pages 660–672, 2012. doi:10.1007/978-3-642-31585-5\_58.
- 54 David Peleg and Vitaly Rubinovich. A near-tight lower bound on the time complexity of distributed minimum-weight spanning tree construction. *SIAM J. Comput.*, 30(5):1427–1442, 2000. doi:10.1137/S0097539700369740.
- 55 A.A. Razborov. On the distributional complexity of disjointness. *Theoretical Computer Science*, 106(2):385–390, 1992. doi:10.1016/0304-3975(92)90260-M.
- 56 Virginia Vassilevska Williams. On some fine-grained questions in algorithms and complexity. In *Proceedings of the International Congress of Mathematicians (ICM 2018)*, pages 3447–3487, 2018. doi:10.1142/9789813272880\_0188.
- 57 Andrew Chi-Chih Yao. On constructing minimum spanning trees in k-dimensional spaces and related problems. *SIAM J. Comput.*, 11(4):721–736, 1982. doi:10.1137/0211059.





# Fast Hybrid Network Algorithms for Shortest Paths in Sparse Graphs

**Michael Feldmann**

Universität Paderborn, Germany  
michael.feldmann@upb.de

**Kristian Hinnenthal**

Universität Paderborn, Germany  
krijan@mail.upb.de

**Christian Scheideler**

Universität Paderborn, Germany  
scheideler@upb.de

---

## Abstract

We consider the problem of computing shortest paths in *hybrid networks*, in which nodes can make use of different communication modes. For example, mobile phones may use ad-hoc connections via Bluetooth or Wi-Fi in addition to the cellular network to solve tasks more efficiently. Like in this case, the different communication modes may differ considerably in range, bandwidth, and flexibility. We build upon the model of Augustine et al. [SODA '20], which captures these differences by a *local* and a *global* mode. Specifically, the local edges model a fixed communication network in which  $O(1)$  messages of size  $O(\log n)$  can be sent over every edge in each synchronous round. The global edges form a clique, but nodes are only allowed to send and receive a total of at most  $O(\log n)$  messages over global edges, which restricts the nodes to use these edges only very sparsely.

We demonstrate the power of hybrid networks by presenting algorithms to compute Single-Source Shortest Paths and the diameter very efficiently in *sparse graphs*. Specifically, we present exact  $O(\log n)$  time algorithms for cactus graphs (i.e., graphs in which each edge is contained in at most one cycle), and 3-approximations for graphs that have at most  $n + O(n^{1/3})$  edges and arboricity  $O(\log n)$ . For these graph classes, our algorithms provide exponentially faster solutions than the best known algorithms for general graphs in this model. Beyond shortest paths, we also provide a variety of useful tools and techniques for hybrid networks, which may be of independent interest.

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

**Keywords and phrases** hybrid networks, overlay networks, sparse graphs, cactus graphs

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2020.31

**Related Version** A full version of the paper is available at <https://arxiv.org/abs/2007.01191>.

**Funding** This work is supported by the German Research Foundation (DFG) within the CRC 901 "On-The-Fly Computing" (project number 160364472-SFB901).

## 1 Introduction

The idea of *hybrid networks* is to leverage multiple communication modes with different characteristics to deliver scalable throughput, or to reduce complexity, cost or power consumption. In *hybrid data center networks* [10], for example, the server racks can make use of optical switches [13] or wireless antennas [11] to establish direct connections in addition to using the traditional electronic packet switches. Other examples of hybrid communication are combining multipoint with standard VPN connections [30], hybrid WANs [32], or mobile phones using device-to-device communication in addition to cellular networks as in 5G [22]. As a consequence, several theoretical models and algorithms have been proposed for hybrid networks in recent years [16, 20, 4, 5].



© Michael Feldmann, Kristian Hinnenthal, and Christian Scheideler;  
licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 31; pp. 31:1–31:16

Leibniz International Proceedings in Informatics



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



In this paper, we focus on the general hybrid network model of Augustine et al. [5]. The authors distinguish two different modes of communication, a *local* mode, which nodes can use to send messages to their neighbors in an input graph  $G$ , and a *global* mode, which allows the nodes to communicate with *any* other node of  $G$ . The model is parameterized by the number of messages  $\lambda$  that can be sent over each local edge in each round, and the total number of messages  $\gamma$  that each node can send and receive over global edges in a single round. Therefore, the local network rather relates to *physical* networks, where an edge corresponds to a dedicated connection that cannot be adapted by the nodes, e.g., a cable, an optical connection, or a wireless ad-hoc connection. On the other hand, the global network captures characteristics of *logical* networks, which are formed as overlays of a shared physical infrastructure such as the internet or a cellular network. Here, nodes can in principle contact any other node, but can only perform a limited amount of communication in each round.

Specifically, we consider the hybrid network model with  $\lambda = O(1)$  and  $\gamma = O(\log n)$ , i.e., the local network corresponds to the CONGEST model [28], whereas the global network is the so-called *node-capacitated clique* (NCC) [4, 1, 29]. Thereby, we only grant the nodes very limited communication capabilities for both communication modes, disallowing them, for example, to gather complete neighborhood information to support their computation. With the exception of a constant factor SSSP approximation, none of the shortest paths algorithms of [5], for example, can be directly applied to this very restricted setting, since [5] assumes the LOCAL model for the local network. Furthermore, our algorithms do not even exploit the power of the NCC for the global network; in fact, they would also work if the nodes would initially only knew their neighbors in  $G$  and had to learn new node identifiers via introduction (which has recently been termed the  $NCC_0$  model [3]).

As in [5], we focus on *shortest paths problems*. However, instead of investigating general graphs, we present polylogarithmic time algorithms to compute Single-Source Shortest Paths (SSSP) and the diameter in *sparse graphs*. Specifically, we present randomized  $O(\log n)$  time algorithms for *cactus graphs*, which are graphs in which any two cycles share at most one node. Cactus graphs are relevant for wireless communication networks, where they can model combinations of star/tree and ring networks (e.g., [9]), or combinations of ring and bus structures in LANs (e.g., [24]). However, research on solving graph problems in cactus graphs mostly focuses on the sequential setting.

Furthermore, we present 3-approximate randomized algorithms with runtime  $O(\log^2 n)$  for graphs that contain at most  $n + O(n^{1/3})$  edges and have arboricity<sup>1</sup>  $O(\log n)$ . Graphs with bounded arboricity, which include important graph families such as planar graphs, graphs with bounded treewidth, or graphs that exclude a fixed minor, have been extensively studied in the past years. Note that although these graphs are very sparse, in contrast to cactus graphs they may still contain a polynomial number of (potentially nested) cycles. Our algorithms are exponentially faster than the best known algorithms for general graphs for shortest paths problems [4, 23].

For the *All-Pairs Shortest Paths* (APSP) problem, which is not studied in this paper, there is a lower bound of  $\tilde{\Omega}(\sqrt{n})$  [5, Theorem 2.5] that even holds for  $\tilde{O}(\sqrt{n})$ -approximations<sup>2</sup>. Recently, this lower bound was shown to be tight up to polylogarithmic factors [23]. The bound specifically also holds for trees, which, together with the results in this paper, shows an exponential gap between computing the diameter and solving APSP in trees. Furthermore,

---

<sup>1</sup> The arboricity of a graph  $G$  is the minimum number of forests into which its edges can be partitioned.

<sup>2</sup> The  $\tilde{O}$ -notation hides polylogarithmic factors.

the results of [23] show that computing (an approximation of) the diameter in general graphs takes time roughly  $\Omega(n^{1/3})$  (even with unbounded local communication). Therefore, our paper demonstrates that sparse graphs allow for an exponential improvement.

## 1.1 Model and Problem Definition

We consider a *hybrid network model* in which we are given a fixed node set  $V$  consisting of  $n$  nodes that are connected via *local* and *global* edges. The local edges form a fixed, undirected, and weighted graph  $G = (V, E, w)$  (the *local network*), where the edge weights are given by  $w : E \rightarrow \{1, \dots, W\} \subset \mathbb{N}$  and  $W$  is assumed to be polynomial in  $n$ . We denote the degree of a node  $v$  in the local network by  $\deg(v)$ . Furthermore, every two nodes  $u, v \in V$  are connected via a global edge, i.e., the *global network* forms a clique. Every node  $v \in V$  has a unique identifier  $\text{id}(v)$  of size  $O(\log n)$ , and, since the nodes form a clique in the global network, every node knows the identifier of every other node. Although this seems to be a fairly strong assumption, our algorithms would also work in the  $\text{NCC}_0$  model [3] for the global network, in which each node initially only knows the identifiers of its neighbors in  $G$ , and new connections need to be established by sending node identifiers (which is very similar to the overlay network models of [16, 6, 17]). We further assume that the nodes know  $n$  (or an upper bound polynomial in  $n$ ).

We assume a synchronous message passing model, where in each round every node can send messages of size  $O(\log n)$  over both local and global edges. Messages that are sent in round  $i$  are collectively received at the beginning of round  $i + 1$ . However, we impose different communication restrictions on the two network types. Specifically, every node can send  $O(1)$  (distinct) messages over each of its incident local edges, which corresponds to the **CONGEST** model for the local network [28]. Additionally, it can send and receive at most  $O(\log n)$  many messages over global edges (where, if more than  $O(\log n)$  messages are sent to a node, an arbitrary subset of the messages is delivered), which corresponds to the **NCC** model [4]. Therefore, our hybrid network model is precisely the model proposed in [5] for parameters  $\lambda = O(1)$  and  $\gamma = O(\log n)$ . Note that whereas [5] focuses on the much more generous **LOCAL** model for the local network, our algorithms do not require nor easily benefit from the power of unbounded communication over local edges.

We define the *length* of a path  $P \subseteq E$  as  $w(P) := \sum_{e \in P} w(e)$ . A path  $P$  from  $u$  to  $v$  is a *shortest path*, if there is no path  $P'$  from  $u$  and  $v$  with  $w(P') < w(P)$ . The *distance* between two nodes  $u$  and  $v$  is defined as  $d(u, v) := w(P)$ , where  $P$  is a shortest path from  $u$  to  $v$ .

In the *Single-Source Shortest Paths Problem* (SSSP), there is one node  $s \in V$  and every node  $v \in V$  wants to compute  $d(s, v)$ . In the *Diameter Problem*, every node wants to learn the *diameter*  $D := \max_{u, v \in V} d(u, v)$ . An algorithm computes an  $\alpha$ -approximation of SSSP, if every node  $v \in V$  learns an estimate  $\tilde{d}(s, v)$  such that  $d(s, v) \leq \tilde{d}(s, v) \leq \alpha \cdot d(s, v)$ . Similarly, for an  $\alpha$ -approximation of the diameter, every node  $v \in V$  has to compute an estimate  $\tilde{D}$  such that  $D \leq \tilde{D} \leq \alpha \cdot D$ .

## 1.2 Contribution and Structure of the Paper

The first part of the paper revolves around computing SSSP and the diameter on cactus graphs (i.e., connected graphs in which each edge is only contained in at most one cycle). For a more comprehensive presentation, we establish the algorithm in several steps. First, we consider the problems in path graphs (i.e., connected graphs that contain exactly two nodes with degree 1, and every other node has degree 2; see Section 2), then in cycle graphs (i.e., connected graphs in which each node has degree 2, see Section 3), trees (Section 4), and

pseudotrees (Section 5), which are graphs that contain at most one cycle. For each of these graph classes, we present deterministic algorithms to solve both problems in  $O(\log n)$  rounds, each relying heavily on the results of the previous sections. We then extend our results to cactus graphs (Section 6) and present randomized algorithms for SSSP and the diameter with a runtime of  $O(\log n)$ , w.h.p.<sup>3</sup>

In Section 7, we consider a more general class of sparse graphs, namely graphs with at most  $n + O(n^{1/3})$  edges and arboricity  $O(\log n)$ . By using the techniques established in the first part and leveraging the power of the global network to deal with the additional  $O(n^{1/3})$  edges, we obtain algorithms to compute 3-approximations for SSSP and the diameter in time  $O(\log^2 n)$ , w.h.p. As a byproduct, we also derive a deterministic  $O(\log^2 n)$ -round algorithm for computing a (balanced) hierarchical tree decomposition of the network.

We remark that our algorithms heavily use techniques from the PRAM literature. For example, *pointer jumping* [19], and the *Euler tour* technique (e.g., [31, 2]), which extends pointer jumping to certain graphs such as trees, have been known for decades, and are also used in distributed algorithms (e.g., [16, 6]). As already pointed out in [4], the NCC in particular has a very close connection to PRAMs. In fact, if  $G$  is very sparse, PRAM algorithms can efficiently be simulated in our model even if the edges are very unevenly distributed (i.e., nodes have a very high degree). We formally prove this in the full version of this paper [14]. This allows us to obtain some of our algorithms for path graphs, cycle graphs, and trees by PRAM simulations (see Section 1.3). We nonetheless present our distributed solutions without using PRAM simulations, since (1) a direct simulation only yields randomized algorithms, (2) the algorithms of the later sections heavily build on the basic algorithms of the first sections, (3) a simulation exploits the capabilities of the global network more than necessary. As already pointed out, *all* of our algorithms would also work in the weaker  $NCC_0$  model for the global network, or if the nodes could only contact  $\Theta(\log n)$  random nodes in each round.<sup>4</sup> Furthermore, if we restrict the degree of  $G$  to be  $O(\log n)$ , our algorithms can be modified to run in the  $NCC_0$  without using the local network.

Beyond the results for sparse graphs, this paper contains a variety of useful tools and results for hybrid networks in general, such as Euler tour and pointer jumping techniques for computation in trees, a simple load-balancing framework for low-arboricity graphs, an extension of the recent result of Götte et al. [18] to compute spanning trees in the  $NCC_0$ , and a technique to perform matrix multiplication. In combination with sparse spanner constructions (see, e.g., [8]) or skeletons (e.g., [33]), our algorithms may lead to efficient shortest path algorithms in more general graph classes. Also, our algorithm to construct a hierarchical tree decomposition may be of independent interest, as such constructions are used for example in routing algorithms for wireless networks (see, e.g., [15, 21]).

Due to space constraints, all proofs and figures, as well as the detailed description and some lemmas of our algorithms, are deferred to the full version of this paper [14].

### 1.3 Further Related Work

As theoretical models for hybrid networks have only been proposed recently, only few results for such models are known at this point [16, 4, 5]. Computing an exact solution for SSSP in arbitrary graphs can be done in  $\tilde{O}(\sqrt{SPD})$  rounds [5], where  $SPD$  is the so-called

<sup>3</sup> An event holds with high probability (w.h.p.) if it holds with probability at least  $1 - 1/n^c$  for an arbitrary but fixed constant  $c > 0$ .

<sup>4</sup> We remark that for the algorithms in Section 7 this requires to setup a suitable overlay network like a butterfly in time  $O(\log^2 n)$ , which can be done using well-known techniques.

*shortest path diameter* of  $G$ . For large SPD, this bound has recently been improved to  $\tilde{O}(n^{2/5})$  [23]. The authors of [5] also present several approximation algorithms for SSSP: A  $(1+\varepsilon)$ -approximation with runtime  $\tilde{O}(n^{1/3}/\varepsilon^6)$ , a  $(1/\varepsilon)^{O(1/\varepsilon)}$ -approximation running in  $\tilde{O}(n^\varepsilon)$  rounds and a  $2^{O(\sqrt{\log n \log \log n})}$ -approximation with runtime  $2^{O(\sqrt{\log n \log \log n})}$ . For APSP there is an exact algorithm that runs in  $\tilde{O}(n^{2/3})$  rounds, a  $(1+\varepsilon)$ -approximation running in  $\tilde{O}(\sqrt{n}/\varepsilon)$  rounds (only for unweighted graphs) and a 3-approximation with runtime  $\tilde{O}(\sqrt{n})$  [5]. In [23], the authors give a lower bound of  $\tilde{\Omega}(n^{1/3})$  rounds for computing the diameter in arbitrary graphs in our model. They also give approximation algorithms with approximation factors  $(3/2+\varepsilon)$  and  $(1+\varepsilon)$  that run in time  $\tilde{O}(n^{1/3}/\varepsilon)$  and  $\tilde{O}(n^{0.397}/\varepsilon)$ , respectively. Even though APSP and the diameter problem are closely related, we demonstrate that the diameter can be computed much faster in our hybrid network model for certain graphs classes.

As already pointed out, the global network in our model has a close connection to overlay networks. The NCC model, which has been introduced in [4], mainly focuses on the impact of node capacities, especially when the nodes have a high degree. Since, intuitively, for many graph problems the existence of *each* edge is relevant for the output, most algorithms in [4] depend on the arboricity  $a$  of  $G$  (which is, roughly speaking, the time needed to efficiently distribute the load of all edges over the network). The authors present  $\tilde{O}(a)$  algorithms for local problems such as MIS, matching, or coloring, an  $\tilde{O}(D+a)$  algorithm for BFS tree, and an  $\tilde{O}(1)$  algorithm to compute a minimum spanning tree (MST). Recently,  $\tilde{O}(\Delta)$ -time algorithms for graph realization problems have been presented [3], where  $\Delta$  is the maximum node degree; notably, most of the algorithms work in the  $\text{NCC}_0$  variant. Furthermore, Robinson [29] investigates the information the nodes need to learn to jointly solve graph problems and derives a lower bound for constructing spanners in the NCC. For example, his result implies that spanners with constant stretch require polynomial time in the NCC, and are therefore harder to compute than MSTs. Since our global network behaves like an overlay network, we can make efficient use of the so-called *shortest-path diameter reduction technique* [26]. By adding shortcuts between nodes in the global network, we can bridge large distances quickly throughout our computations.

As argued before, we could apply some of the algorithms for PRAMs to our model instead of using native distributed solutions by using PRAM simulations. For example, we are able to use the algorithms of [12] to solve SSSP and diameter in trees in time  $O(\log n)$ , w.h.p. Furthermore, we can compute the distance between any pair  $s$  and  $t$  in *outerplanar graphs* in time  $O(\log^3 n)$  by simulating a CREW PRAM. For planar graphs, the distance between  $s$  and  $t$  can be computed in time  $O(\log^3 n(1+M(q))/n)$ , w.h.p., where the nodes know a set of  $q$  faces of a planar embedding that covers all vertices, and  $M(q)$  is the number of processors required to multiply two  $q \times q$  matrices in  $O(\log q)$  time in the CREW PRAM.

For graphs with polylogarithmic arboricity, a  $(1+\varepsilon)$ -approximation of SSSP can be computed in polylog time using [25] and our simulation framework (with huge polylogarithmic terms). For general graphs, the algorithm can be combined with well-known spanner algorithms for the CONGEST model (e.g., [8]) to achieve constant approximations for SSSP in time  $\tilde{O}(n^\varepsilon)$  time in our hybrid model. This yields an alternative to the SSSP approximation of [5], which also requires time  $\tilde{O}(n^\varepsilon)$  but has much smaller polylogarithmic factors.

## 2 Path Graphs

To begin with an easy example, we first present a simple algorithm to compute SSSP and the diameter of path graphs. The simple idea of our algorithms is to use *pointer jumping* to select a subset of global edges  $S$ , which we call *shortcut edges*, with the following properties:

$S$  is a weighted connected graph with degree  $O(\log n)$  that contains all nodes of  $V$ , and for every  $u, v \in V$  there exists a path  $P \subseteq S$ ,  $|P| = O(\log n)$  (where  $|P|$  denotes the number of edges of  $P$ ), such that  $w(P) = d(u, v)$ , and no path  $P$  such that  $w(P) < d(u, v)$ . Given such a graph, SSSP can easily be solved by performing a broadcast from  $s$  in  $S$  for  $O(\log n)$  rounds: In the first round,  $s$  sends a message containing  $w(e)$  over each edge  $e \in S$  incident to  $s$ . In every subsequent round, every node  $v \in V$  that has already received a message sends a message  $k + w(e)$  over each edge  $e \in S$  incident to  $v$ , where  $k$  is the smallest value  $v$  has received so far. After  $O(\log n)$  rounds, every node  $v$  must have received  $d(s, v)$ , and cannot have received any smaller value. Further, the diameter of the line can easily be determined by performing SSSP from both of its endpoints  $u, v$ , which finally broadcast the diameter  $d(u, v)$  to all nodes using the global network.

We construct  $S$  using the following simple *Introduction Algorithm*.  $S$  initially contains all edges of  $E$ . Additional shortcut edges are established by performing *pointer jumping*: Every node  $v$  first selects one of its at most two neighbors as its *left neighbor*  $\ell_1$ ; if it has two neighbors, the other is selected as  $v$ 's *right neighbor*  $r_1$ . In the first round of our algorithm, every node  $v$  with degree 2 establishes  $\{\ell_1, r_1\}$  as a new shortcut edge of weight  $w(\{\ell_1, r_1\}) = w(\{\ell_1, v\}) + w(\{v, r_1\})$  by sending the edge to both  $\ell_1$  and  $r_1$ . Whenever at the beginning of some round  $i > 1$  a node  $v$  with degree 2 receives shortcut edges  $\{u, v\}$  and  $\{v, w\}$  from  $\ell_{i-1}$  and  $r_{i-1}$ , respectively, it sets  $\ell_i := u$ ,  $r_i := w$ , and establishes  $\{\ell_i, r_i\}$  by adding up the weights of the two received edges and informing  $\ell_i$  and  $r_i$ . The algorithm terminates after  $\lceil \log(n-1) \rceil$  rounds. Afterwards, for every simple path in  $G$  between  $u$  and  $v$  with  $2^k$  hops for any  $k \leq \lceil \log(n-1) \rceil$  we have established a shortcut edge  $e \in S$  with  $w(e) = d(u, v)$ . Therefore,  $S$  has the desired properties, and we conclude the following theorem.

► **Theorem 1.** *SSSP and the diameter can be computed in any path graph in time  $O(\log n)$ .*

### 3 Cycle Graphs

In cycle graphs, there are two paths between any two nodes that we need to distinguish. For SSSP, this can easily be achieved by performing the SSSP algorithm for path graphs in both directions along the cycle, and let each node choose the minimum of its two computed distances. Formally, let  $v_1, v_2, \dots, v_n$  denote the  $n$  nodes along a *left* traversal of the cycle starting from  $s = v_1$  and continuing at  $s$ 's neighbor of smaller identifier, i.e.,  $\text{id}(v_2) < \text{id}(v_n)$ . For any node  $u$ , a shortest path from  $s$  to  $u$  must follow a left or right traversal along the cycle, i.e.,  $(v_1, v_2, \dots, u)$  or  $(v_1, v_n, \dots, u)$  is a shortest path from  $s$  to  $u$ . Therefore, we can solve SSSP on the cycle by performing the SSSP algorithm for the path graph on  $\mathcal{L} := (v_1, v_2, \dots, v_n)$  and  $\mathcal{R} := (v_1, v_n, v_{n-1}, \dots, v_2)$ . Thereby, every node  $v$  learns  $d_\ell(s, v)$ , which is the distance from  $s$  to  $v$  in  $\mathcal{L}$  (i.e., along a left traversal of the cycle), and  $d_r(s, v)$ , which is their distance in  $\mathcal{R}$ . It is easy to see that  $d(s, v) = \min\{d_\ell(s, v), d_r(s, v)\}$ .

Using the above algorithm,  $s$  can also easily learn its *eccentricity*  $\text{ecc}(s) := \max_{v \in V} \{d(s, v)\}$ , as well as its *left and right farthest nodes*  $s_\ell$  and  $s_r$ . The left farthest node  $s_\ell$  of  $s$  is defined as the farthest node  $v_i$  along a left traversal of the cycle such that the subpath in  $\mathcal{L}$  from  $s = v_1$  to  $v_i$  is still a shortest path. Formally,  $s_\ell = \arg \max_{v \in V, d_\ell(s, v) \leq \lfloor W/2 \rfloor} d_\ell(s, v)$ , where  $W = \sum_{e \in E} w(e)$ . The right farthest node  $s_r$  is the successor of  $s_\ell$  in  $\mathcal{L}$  (or  $s$ , if  $s_\ell$  is the last node of  $\mathcal{L}$ ), for which it must hold that  $d_r(s, s_r) \leq \lfloor W/2 \rfloor$ . Note that  $d_\ell(s, s_\ell) = d(s, s_\ell)$ ,  $d_r(s, s_r) = d(s, s_r)$ , and  $\text{ecc}(s) = \max\{d_\ell(s, s_\ell), d_r(s, s_r)\}$ .

To determine the diameter of  $G$ , for every node  $v \in V$  our goal is to compute  $\text{ecc}(v)$ ; as a byproduct, we will compute  $v$ 's left and right farthest nodes  $v_\ell$  and  $v_r$ . The diameter can then be computed as  $\max_{v \in V} \text{ecc}(v)$ . A simple way to compute these values is to employ a

binary-search style approach from all nodes in parallel, and use load balancing techniques from [4] to achieve a runtime of  $O(\log^2 n)$ , w.h.p. Coming up with a deterministic  $O(\log n)$  time algorithm, however, is more complicated.

Due to space constraints, we defer the description of the algorithm to the full version.

► **Theorem 2.** *SSSP and the diameter can be computed in any cycle graph  $G$  in time  $O(\log n)$ .*

## 4 Trees

We now show how the algorithms of the previous sections can be extended to compute SSSP and the diameter on trees. As in the algorithm of Gmyr et al. [16], we adapt the well-known *Euler tour* technique to a distributed setting and transform the graph into a path  $L$  of *virtual nodes* that corresponds to a depth-first traversal of  $G$ . More specifically, every node of  $G$  simulates one virtual node for each time it is visited in that traversal, and two virtual nodes are neighbors in  $L$  if they correspond to subsequent visitations. To solve SSSP, we assign weights to the edges from which the initial distances in  $G$  can be inferred, and then solve SSSP in  $L$  instead. Finally, we compute the diameter of  $G$  by performing the SSSP algorithm twice, which concludes this section.

However, since a node can be visited up to  $\Omega(n)$  times in the traversal, it may not be able to simulate all of its virtual nodes in  $L$ . Therefore, we first need to reassign the virtual nodes to the node's neighbors such that every node only has to simulate at most 6 virtual nodes using the *Nash-Williams forests decomposition* technique [27]. More precisely, we compute an *orientation* of the edges in which each node has outdegree at most 3, and reassign nodes according to this orientation (in the remainder of this paper, we refer to this as the *redistribution framework*). Due to space constraints, we defer a precise description of the algorithm to the full version and only state our main results.

The following two lemmas follow from applying PRAM techniques.

► **Lemma 3.** *Let  $H = (V, E)$  be a forest in which every node  $v \in V$  stores some value  $p_v$ , and let  $f$  be a distributive aggregate function<sup>5</sup>. Every node  $v \in V$  can learn  $f(\{p_u \mid u \in C_v\})$ , where  $C_v$  is the tree of  $H$  that contains  $v$ , in time  $O(\log n)$ .*

► **Lemma 4.** *Any tree  $G$  can be rooted in  $O(\log n)$  time.*

By assigning positive or negative weights to the edges of  $L$  according to their direction in the rooted version of  $G$ , we easily obtain the following theorem.

► **Theorem 5.** *SSSP can be computed in any tree in time  $O(\log n)$ .*

Similar techniques lead to the following lemmas, which we will use in later sections.

► **Lemma 6.** *Let  $H = (V, E)$  be a forest and assume that each node  $v \in V$  stores some value  $p_v$ . The goal of each node  $v$  is to compute the value  $\text{sum}_v(u) := \sum_{w \in C_u} p_w$  for each of its neighbors  $u$ , where  $C_u$  is the connected component  $C$  of the subgraph  $H'$  of  $H$  induced by  $V \setminus \{v\}$  that contains  $u$ . The problem can be solved in time  $O(\log n)$ .*

<sup>5</sup> 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.



► **Lemma 7.** *Let  $G$  be a tree rooted at  $s$ . Every node  $v \in V$  can compute its height  $h(v)$  in  $G$ , which is length of the longest path from  $v$  to any leaf in its subtree, in time  $O(\log n)$ .*

For the diameter, we use the following well-known lemma.

► **Lemma 8.** *Let  $G$  be a tree,  $s \in V$  be an arbitrary node, and let  $v \in V$  such that  $d(s, v)$  is maximal. Then  $\text{ecc}(v) = D$ .*

Therefore, for the diameter it suffices to perform SSSP once from the node  $s$  with highest identifier, then choose a node  $v$  with maximum distance to  $s$ , and perform SSSP from  $v$ . Since  $\text{ecc}(v) = D$ , the node with maximum distance to  $v$  yields the diameter. Together with Lemma 3, we conclude the following theorem.

► **Theorem 9.** *The diameter can be computed in any tree in time  $O(\log n)$ .*

## 5 Pseudotrees

Recall that a pseudotree is a graph that contains at most one cycle. We define a *cycle node* to be a node that is part of a cycle, and all other nodes as *tree nodes*. For each cycle node  $v$ , we define  $v$ 's tree  $T_v$  as the connected component that contains  $v$  in the graph in which  $v$ 's two adjacent cycle nodes are removed, and denote  $h(v)$  as the height of  $v$  in  $T_v$ . Due to space constraints, we omit the details of the algorithm for pseudotrees and only give a brief description. To compute SSSP, we first need to distinguish the cycle nodes from the tree nodes. We do this by establishing rings of virtual nodes using the approach of Section 4 (which must create two rings in a pseudotree). Then, we can reduce the problem to computing SSSP in cycles and trees, for which we use the algorithms from the previous sections.

► **Theorem 10.** *SSSP can be computed in any pseudotree in time  $O(\log n)$ .*

Computing the diameter is more complicated. Since the longest path may not use any cycle node at all, each cycle node  $v$  first contributes the diameter of its tree  $T_v$  as a possible candidate. Furthermore,  $v$  needs to compute its eccentricity  $\text{ecc}(v)$ , and, if its eccentricity is larger than the height  $h(v)$  of its tree  $T_v$ , contribute  $\text{ecc}(v) + h(v)$ . To compute its eccentricity, every cycle node needs to compute the distance to its farthest nodes using the algorithm of Theorem 2, but also take into account the heights of the trees on the path to these nodes (as a longer path may lead into those trees).

► **Theorem 11.** *The diameter can be computed in any pseudotree in time  $O(\log n)$ .*

## 6 Cactus Graphs

Our algorithm for cactus graphs relies on an algorithm to compute the maximal biconnected components (or *blocks*) of  $G$ , where a graph is called biconnected if the removal of a single node would not disconnect the graph. Note that for any graph, each edge lies in exactly one block. In case of cactus graphs, each block is either a single edge or a simple cycle. By computing the blocks of  $G$ , each node  $v \in V$  classifies its incident edges into bridges (if there is no other edge incident to  $v$  contained in the same block) and pairs of edges that lie in the same cycle. To do so, we first give a variant of [18, Theorem 1.3] for the  $\text{NCC}_0$  under the constraint that the input graph (which is not necessarily a cactus graph) has constant degree. We point out how the lemma is helpful for cactus graphs, and then use a simulation of the biconnectivity algorithm of [31] as in [18, Theorem 1.4] to compute the blocks of  $G$ . The description and proofs of the following three lemmas are very technical and mainly describe adaptations of [18]. Therefore, we defer them to the full version [14].



► **Lemma 12** (Variant of [18, Theorem 1.3]). *Let  $G$  be any graph with constant degree. A spanning tree of  $G$  can be computed in time  $O(\log n)$ , w.h.p., in the  $NCC_0$ .*

► **Lemma 13.** *A spanning tree of a cactus graph  $G$  can be computed in time  $O(\log n)$ , w.h.p.*

► **Lemma 14.** *The biconnected components of a cactus graph  $G$  can be computed in time  $O(\log n)$ , w.h.p.*

Thus, every node can determine which of its incident edges lie in the same block in time  $O(\log n)$ , w.h.p. Let  $s$  be the source for the SSSP problem. First, we compute the *anchor node* of each cycle in  $G$ , which is the node of the cycle that is closest to  $s$  (if  $s$  is a cycle node, then the anchor node of that cycle is  $s$  itself). To do so, we replace each cycle  $C$  in  $G$  by a binary tree  $T_C$  of height  $O(\log n)$  as described in [16]. More precisely, we first establish shortcut edges using the Introduction algorithm in each cycle, and then perform a broadcast from the node with highest identifier in  $C$  for  $O(\log n)$  rounds. If in some round a node receives the broadcast for the first time from  $\ell_i$  or  $r_i$ , it sets that node as its parent in  $T_C$  and forwards the broadcast to  $\ell_j$  and  $r_j$ , where  $j = \min\{i - 1, 0\}$ . After  $O(\log n)$  rounds,  $T_C$  is a binary tree that contains all nodes of  $C$  and has height  $O(\log n)$ . To perform the execution in all cycles in parallel, each node simulates one virtual node for each cycle it lies in and connects the virtual nodes using their knowledge of the blocks of  $G$ . To keep the global communication low, we again use the redistribution framework described in Section 4 (note that the arboricity of  $G$  is 2).

► **Lemma 15.** *Let  $T$  be the (unweighted) tree that results from taking the union of all trees  $T_C$  and all bridges in  $G$ . For each cycle  $C$ , the node  $a_C := \arg \min_{v \in C} d_T(s, v)$  is the anchor node of  $C$ .*

The correctness of the lemma above simply follows from the fact that any shortest path from  $s$  to any node in  $C$  must contain the anchor node of  $C$  both in  $G$  and in  $T$ . Therefore, the anchor node of each cycle can be computed by first performing the SSSP algorithm for trees with source  $s$  in  $T$  and then conducting a broadcast in each cycle. Now let  $v$  be the anchor node of some cycle  $C$  in  $G$ . By performing the diameter algorithm of Theorem 2 in  $C$ ,  $v$  can compute its left and right farthest nodes  $v_\ell$  and  $v_r$  in  $C$ . Again, to perform all executions in parallel, we use our redistribution framework.

► **Lemma 16.** *Let  $S_G$  be the graph that results from removing the edge  $\{v_\ell, v_r\}$  from each cycle  $C$  with anchor node  $v$ .  $S_G$  is a shortest path tree of  $G$  with source  $s$ .*

Therefore, we can perform the SSSP algorithm for trees of Theorem 5 on  $S_G$  and obtain the following theorem.

► **Theorem 17.** *SSSP can be computed in any cactus graph in time  $O(\log n)$ .*

To compute the diameter, we first perform the algorithm of Lemma 16 with the node that has highest identifier as source  $s$ ,<sup>6</sup> which yields a shortest path tree  $S_G$ . This tree can easily be rooted using Lemma 4. Let  $Q(v)$  denote the children of  $v$  in  $S_G$ , and let  $B(v)$  denote the block of node  $v$  in  $G$ . Using Lemma 7, each node  $v$  can compute its height  $h(v)$  in  $S_G$  and can locally determine the value  $m(v) := \max_{u, w \in Q(v), B(u) \neq B(w)} (h(u) + h(w) + w(v, u) + w(v, w))$ . We further define the pseudotree  $\Pi_C$  of each cycle  $C$  as the graph that contains all edges of

<sup>6</sup> In the  $NCC_0$ , this node can be determined by constructing the tree  $T$  from Lemma 15 and using Lemma 3 on  $T$ .

$C$  and, additionally, an edge  $\{v, t_v\}$  for each node  $v \neq a_C$  of  $C$ , where  $t_v$  is a node that is simulated by  $v$ , and  $w(\{v, t_v\}) = \max_{u \in Q(v) \setminus C} (h(u) + w(\{v, u\}))$ . Intuitively, each node  $v$  of  $C$  that is not the anchor node is attached an edge whose weight equals the height of its subtree in  $S_G$  without considering the child of  $v$  that also lies in  $C$  (if that exists). Then, for each cycle  $C$  in parallel, we perform the algorithm of Theorem 11 on  $\Pi_C$  to compute its diameter  $D(\Pi_C)$  (using the redistribution framework). We obtain the diameter of  $G$  as the value  $\hat{D} := \max\{\max_{v \in V} (h(v)), \max_{v \in V} (m(v)), \max_{\text{cycle } C} (D(\Pi_C))\}$ . By showing that  $\hat{D} = D$ , we conclude the following theorem.

► **Theorem 18.** *The diameter can be computed in any cactus graph in time  $O(\log n)$ .*

## 7 Sparse Graphs

In this final section, we present constant factor approximations for SSSP and the diameter in graphs that contain at most  $n + O(n^{1/3})$  edges and that have arboricity at most  $O(\log n)$ . Our algorithm for such graphs relies on an MST  $M = (V, E')$  of  $G$ , where  $E' \subseteq E$ .  $M$  can be computed deterministically in time  $O(\log^2 n)$  using [16], Observation 4, in a modified way<sup>7</sup>.

► **Lemma 19.** *The algorithm computes an MST of  $G$  deterministically in time  $O(\log^2 n)$ .*

We call each edge  $e \in E \setminus E'$  a *non-tree edge*. Further, we call a node *shortcut node* if it is adjacent to a non-tree edge, and define  $\Sigma \subseteq V$  as the set of shortcut nodes. Clearly, after computing  $M$  every node  $v \in \Sigma$  knows that it is a shortcut node, i.e., if one of its incident edges has not been added to  $E'$ . In the remainder of this section, we will compute approximate distances by (1) computing the distance from each node to its closest shortcut node in  $G$ , and (2) determining the distance between any two shortcut nodes in  $G$ . For any  $s, t \in V$ , we finally obtain a good approximation for  $d(s, t)$  by considering the path in  $M$  as well as a path that contains the closest shortcut nodes of both  $s$  and  $t$ .

Our algorithms rely on a *balanced decomposition tree*  $T_M$ , which allows us to quickly determine the distance between any two nodes in  $G$ , and which is presented in Section 7.1. In Section 7.2,  $T_M$  is extended by a set of edges that allow us to solve (1) by performing a distributed multi-source Bellman-Ford algorithm for  $O(\log n)$  rounds. For (2), in Section 7.3 we first compute the distance between any two shortcut nodes in  $M$ , and then perform matrix multiplications to obtain the pairwise distances between shortcut nodes in  $G$ . By exploiting the fact that  $|\Sigma| = O(n^{1/3})$ , and using techniques of [4], we are able to distribute the  $\Theta(n)$  operations of each of the  $O(\log n)$  multiplications efficiently using the global network. In Section 7.4, we finally show how the information can be used to compute 3-approximations for SSSP and the diameter.

For simplicity, in the following sections we assume that  $M$  has degree 3. Justifying this assumption, we remark that  $M$  can easily be transformed into such a tree while preserving the distances in  $M$ . First, we root the tree at the node with highest identifier using Lemma 4. Then, every node  $v$  replaces the edges to its children by a binary tree of virtual nodes, where the leaf nodes are the children of  $v$ , the edge from each leaf  $u$  to its parent is assigned the weight  $w(\{v, u\})$ , and all inner edges have weight 0.<sup>8</sup> The virtual nodes are distributed evenly among the children of  $v$  such that each child is only tasked with the simulation of at most one virtual node. Note that the virtual edges can be established using the local network.

<sup>7</sup> The algorithm of [16] computes a (not necessarily minimum) spanning tree, which would actually already suffice for the results of this paper. However, if  $G$  contains edges with exceptionally large weights, an MST may yield much better results in practice.

<sup>8</sup> Note that the edge weights are no longer strictly positive; however, one can easily verify that the algorithms of this section also work with non-negative edge weights.

## 7.1 Hierarchical Tree Decomposition

We next present an algorithm to compute a hierarchical tree decomposition of  $M$ , resulting in a *balanced decomposition tree*  $T_M$ .  $T_M$  will enable us to compute distances between nodes in  $M$  in time  $O(\log n)$ , despite the fact that the diameter of  $M$  may be very high.

Our algorithm constructs  $T_M$  as a binary rooted tree  $T_M = (V, E_T)$  of height  $O(\log n)$  with root  $r \in V$  (which is the node that has highest identifier) by selecting a set of global edges  $E_T$ . Each node  $v \in V$  knows its parent  $p_T(v) \in V$ . To each edge  $\{u, v\} \in E_T$  we assign a weight  $w(\{u, v\})$  that equals the sum of the weights of all edges on the (unique) path from  $u$  to  $v$  in  $M$ . Further, each node  $v \in V$  is assigned a distinct label  $l(v) \in \{0, 1\}^{O(\log n)}$  such that  $l(v)$  is a prefix of  $l(u)$  for all children  $u$  of  $v$  in  $T_M$ , and  $l(r) = \varepsilon$  (the empty word).

From a high level, the algorithm works as follows. Starting with  $M$ , within  $O(\log n)$  iterations  $M$  is divided into smaller and smaller components until each component consists of a single node. More specifically, in iteration  $i$ , every remaining component  $A$  handles one *recursive call* of the algorithm, where each recursive call is performed independently from the recursive calls executed in other components. The goal of  $A$  is to select a *split node*  $x$ , which becomes a node at depth  $i - 1$  in  $T_M$ , and whose removal from  $M$  divides  $A$  into components of size at most  $|A|/2$ . The split node  $x$  then recursively calls the algorithm in each resulting component; the split nodes that are selected in each component become children of  $x$  in  $T_M$ .

When the algorithm is called at some node  $v$ , it is associated with a *label* parameter  $l \in \{0, 1\}^{O(\log n)}$  and a *parent* parameter  $p \in V$ . The first recursive call is initiated at node  $r$  with parameters  $l = \varepsilon$  and  $p = \emptyset$ . Assume that a recursive call is issued at  $v \in V$ , let  $A$  be the component of  $M$  in which  $v$  lies, and let  $A_1, A_2$  and  $A_3$  be the at most three components of  $A$  that result from removing  $v$ . Using Lemma 6, every node  $u$  in  $A_1$  can easily compute the number of nodes that lie in each of its adjacent subtrees in  $A_1$  (i.e., the size of the resulting components of  $A_1$  after removing  $u$ ). It is easy to see that there must be a *split node*  $x_1$  in  $A_1$  whose removal divides  $A_1$  into components of size at most  $|A|/2$  (see, e.g., [5, Lemma 4.1]); if there are multiple such nodes, let  $x_1$  be the one that has highest identifier. Correspondingly, there are split nodes  $x_2$  in  $A_2$  and  $x_3$  in  $A_3$ .  $v$  learns  $x_1, x_2$  and  $x_3$  using Lemma 3 and sets these nodes as its children in  $T_M$ . By performing the SSSP algorithm of Theorem 5 with source  $v$  in  $A_1$ ,  $x_1$  learns  $d_M(x_1, v)$ , which becomes the weights of the edge  $\{v, x_1\}$  (correspondingly, the edges  $\{v, x_2\}$  and  $\{v, x_3\}$  are established). To continue the recursion in  $A_1$ ,  $x$  calls  $x_1$  with label parameter  $l \circ 00$  and parent parameter  $v$ . Correspondingly,  $x_2$  is called with  $l \circ 01$ , and  $x_3$  with  $l \circ 10$ .

► **Theorem 20.** *A balanced decomposition tree  $T_M$  for  $M$  can be computed in time  $O(\log^2 n)$ .*

It is easy to see that one can route a message from any node  $s$  to any node  $t$  in  $O(\log n)$  rounds by following the unique path in the tree from  $s$  to  $t$ , using the node labels to find the next node on the path. However, the sum of the edge's weights along that path may be higher than the actual distance between  $s$  and  $t$  in  $M$ .

## 7.2 Finding Nearest Shortcut Nodes

To efficiently compute the nearest shortcut node for each node  $u \in V$ , we extend  $T_M$  to a *distance graph*  $D_T = (V, E_D)$ ,  $E_D \supseteq E_T$ , by establishing additional edges between the nodes of  $T_M$ . Specifically, unlike  $T_M$ , the distance between any two nodes in  $D_T$  will be equal to their distance in  $M$ , which allows us to employ a distributed Bellman-Ford approach.

We describe the algorithm to construct  $D_T$  from the perspective of a fixed node  $u \in V$ . For each edge  $\{u, v\} \in E_T$  such that  $u = p_T(v)$  for which there does *not* exist a local edge  $\{u, v\} \in E'$ , we know that the edge  $\{u, v\}$  “skips” the nodes on the unique path between

## 31:12 Fast Hybrid Network Algorithms for Shortest Paths in Sparse Graphs

$u$  and  $v$  in  $M$ . Consequently, these nodes must lie in a subtree of  $v$  in  $T_M$ . Therefore, to compute the exact distance from  $u$  to a skipped node  $w$ , we cannot just simply add up the edges in  $E_T$  on the path from  $u$  to  $w$ , as this sum must be larger than the distance  $d(u, w)$ .

To circumvent this problem,  $u$ 's goal is to establish additional edges to some of these skipped nodes. Let  $x \in V$  be the neighbor of  $u$  in  $M$  that lies on the unique path from  $u$  to  $v$  in  $M$ . To initiate the construction of edges in each of its subtrees,  $u$  needs to send messages to *each* child  $v$  in  $T_M$  that skipped some nodes (recall that  $u$  is able to do so because it has degree 3 in  $T_M$ ). Such a message to  $v$  contains  $l(x)$ ,  $l(u)$ ,  $\text{id}(u)$  and  $w(\{u, v\})$ . Upon receiving the call from  $u$ ,  $v$  contacts its child node  $y$  in  $T_M$  whose label is a prefix of  $l(x)$ , forwarding  $u$ 's identifier,  $l(x)$  and the (updated) weight  $w(\{y, u\}) = w(\{u, v\}) - w(\{v, y\})$ .  $y$  then adds the edge  $\{y, u\}$  with weight  $w(\{y, u\})$  to the set  $E_D$  by informing  $u$  about it. Then,  $y$  continues the recursion at its child in  $T_M$  that lies in  $x$ 's direction, until the process reaches  $x$  itself. Since the height of  $T_M$  is  $O(\log n)$ ,  $u$  learns at most  $O(\log n)$  additional edges and thus its degree in  $D_T$  is  $O(\log n)$ .

Note that since the process from  $u$  propagates down the tree level by level, we can perform the algorithm at all nodes in parallel, whereby the separate construction processes follow each other in a pipelined fashion without causing too much communication. Together with Theorem 20, we obtain the following lemma.

► **Lemma 21.** *The distance graph  $D_T = (V, E_D)$  for  $M$  can be computed in time  $O(\log^2 n)$ .*

From the way we construct the node's additional edges in  $E_D$ , and the fact that the edges in  $E_T$  preserve distances in  $M$ , we conclude the following lemma.

► **Lemma 22.** *For any edge  $\{u, v\} \in E_D$  it holds  $w(\{u, v\}) = d_M(u, v)$ , where  $d_M(u, v)$  denotes the distance between  $u$  and  $v$  in  $M$ .*

The next lemma is crucial for showing the correctness of the algorithms that follow.

► **Lemma 23.** *For every  $u, v \in V$  we have that (1) every path from  $u$  to  $v$  in  $D_T$  has length at least  $d_M(u, v)$ , and (2) there exists a path  $P$  with  $w(P) = d_M(u, v)$  and  $|P| = O(\log n)$  that only contains nodes of the unique path from  $u$  to  $v$  in  $T_M$ .*

For any node  $v \in V$ , we define the *nearest shortcut node* of  $v$  as  $\sigma(v) = \arg \min_{u \in \Sigma} d(v, u)$ . To let each node  $v$  determine  $\sigma(v)$  and  $d(v, \sigma(v))$ , we perform a distributed version of the Bellman-Ford algorithm. From an abstract level, the algorithm works as follows. In the first round, every shortcut node sends a message associated with its own identifier and distance value 0 to itself. In every subsequent round, every node  $v \in V$  chooses the message with smallest distance value  $d$  received so far (breaking ties by choosing the one associated with the node with highest identifier), and sends a message containing  $d + w(\{v, u\})$  to each neighbor  $u$  in  $D_T$ . After  $O(\log n)$  rounds, every node  $v$  knows the distance  $d_M(v, u)$  to its closest shortcut node  $u$  in  $M$ . Since for any closest shortcut node  $w$  in  $G$  there must be a shortest path from  $v$  to  $w$  that only contains edges of  $M$ , this implies that  $u$  must also be closest to  $v$  in  $G$ , i.e.,  $u = \sigma(v)$ , and  $d_M(v, u) = d(v, \sigma(v))$ .

Note that each node has only created additional edges to its descendants in  $T_M$  during the construction of  $D_T$ , therefore the degree of  $D_T$  is  $O(\log n)$  and we can easily perform the algorithm described above using the global network.

► **Lemma 24.** *After  $O(\log n)$  rounds, each node  $v \in V$  knows  $\text{id}(u)$  of its nearest shortcut node  $\sigma(v)$  in  $G$  and its distance  $d(v, \sigma(v))$  to it.*

### 7.3 Computing APSP between Shortcut Nodes

In this section, we first describe how the shortcut nodes can compute their pairwise distances in  $M$  by using  $D_T$ . Then, we explain how the information can be used to compute all pairwise distances between shortcut nodes in  $G$  by performing matrix multiplications.

**Compute Distances in  $M$ .** First, each node learns the total number of shortcut nodes  $n_c := |\Sigma|$ , and each shortcut node is assigned a unique identifier from  $[n_c]$ .<sup>9</sup> The first part can easily be achieved using Lemma 3. For the second part, consider the Patricia trie  $P$  on the node's identifiers, which, since each node knows all identifiers, is implicitly given to the nodes. By performing a convergecast in  $P$  (where each inner node is simulated by the leaf node in its subtree that has highest identifier), every inner node of  $P$  can learn the number of shortcut nodes in its subtree in  $P$ . This allows the root of  $P$  to assign intervals of labels to its children in  $P$ , which further divide the interval according to the number of shortcut nodes in their children's subtrees, until every shortcut node is assigned a unique identifier.

Note that it is impossible for a shortcut node to explicitly learn all the distances to all other shortcut nodes in polylogarithmic time, since it may have to learn  $\Omega(n^{1/3})$  many bits. However, if we could distribute the distances of all  $O(n^{2/3})$  pairs of shortcut nodes uniformly among all nodes of  $V$ , each node would only have to store  $O(\log n)$  bits<sup>10</sup>. We make use of this in the following way. To each pair  $(i, j)$  of shortcut nodes we assign a *representative*  $h(i, j) \in V$ , which is chosen using (pseudo-)random hash function  $h : [n_c]^2 \rightarrow V$  that is known to all nodes and that satisfies  $h(i, j) = h(j, i)$ .<sup>11</sup> The goal of  $h(i, j)$  is to infer  $d_M(i, j)$  from learning all the edges on the path from  $i$  to  $j$  in  $D_T$ .

Due to space reasons, we defer a precise description to the full version. From a high level, each  $h(i, j)$  first needs to retrieve the labels of both  $i$  and  $j$  in  $T_M$ , which it cannot do directly, as the nodes may be contacted by many other nodes. Instead, we use techniques from [4] to distribute the load:  $h(i, j)$  participates in the construction of a *multicast tree* towards both  $i$  and  $j$ . Using randomization, these trees can be used to disseminate information from each node to all nodes in its multicast tree in a broadcast fashion with low congestion rather than communicating directly. Afterwards,  $h(i, j)$  can infer the labels of all nodes on the path from  $i$  to  $j$  in  $D_M$ , and learn their edge weights in a very similar way. By observing that each node only has to learn  $O(\log n)$  values, we can use Theorems 2.3 and Theorem 2.4 of [4] in a straight-forward manner to obtain the following lemma.

► **Lemma 25.** *Every representative  $h(i, j)$  learns  $d_M(i, j)$  in time  $O(\log n)$ , w.h.p.*

**Compute Distances in  $G$ .** Let  $A \in \mathbb{N}_0^{n_c \times n_c}$  be the *distance matrix* of the shortcut nodes, where  $A_{i,j} = \min\{w(\{i, j\}), d_M(i, j)\}$ , if  $\{i, j\} \in E$ , and  $A_{i,j} = d_M(i, j)$ , otherwise. Our goal is to square  $A$  for  $\lceil \log n \rceil + 2$  many iterations in the *min-plus semiring*. More precisely, we define  $A^1 = A$ , and for  $t \geq 1$  we have that  $A_{i,j}^{2^t} = \min_{k \in [n_c]} (A_{i,k}^{2^{t-1}} + A_{k,j}^{2^{t-1}})$ . The following lemma shows that after squaring the matrix  $\lceil \log n \rceil + 2$  times, its entries give the distances in  $G$ .

<sup>9</sup> We denote  $[k] = \{0, \dots, k-1\}$ .

<sup>10</sup> In fact, for this we could even allow  $n$  pairs, i.e.,  $n_c = O(\sqrt{n})$ ; the reason for our bound on  $n_c$  will become clear later.

<sup>11</sup> Note that sufficient shared randomness can be achieved in our model by broadcasting  $\Theta(\log^2 n)$  random bits in time  $O(\log n)$  [4]. Further, note that for a node  $v \in V$  there can be up to  $O(\log n)$  keys  $(i, j)$  for which  $h(i, j) = v$ , w.h.p., thus  $v$  has to act on behalf of at most  $O(\log n)$  nodes.

► **Lemma 26.**  $A_{i,j}^{2^{\lceil \log n \rceil + 2}} = d(i,j)$  for each  $i, j \in \Sigma$ .

We now describe how the matrix can efficiently be multiplied. As an invariant to our algorithm, we show that at the beginning of the  $t$ -th multiplication, every representative  $h(i, j)$  stores  $A_{i,j}^{2^{t-1}}$ . Thus, for the induction basis we first need to ensure that every representative  $h(i, j)$  learns  $A_{i,j}$ . By Lemma 25,  $h(i, j)$  already knows  $d_M(i, j)$ , thus it only needs to retrieve  $w(\{i, j\})$ , if that edge exists. To do so, we first compute an orientation with outdegree  $O(\log n)$  in time  $O(\log n)$  using [7, Corollary 3.12] in the local network. For every edge  $\{i, j\}$  that is directed from  $i$  to  $j$ ,  $i$  sends a message containing  $w(\{i, j\})$  to  $h(i, j)$ ; since the arboricity of  $G$  is  $O(\log n)$ , every node only has to send at most  $O(\log n)$  messages.

The  $t$ -th multiplication is then done in the following way. We use a (pseudo-)random hash function  $h : [n_c]^3 \rightarrow V$ , where  $h(i, j, k) = h(j, i, k)$ . First, every node  $h(i, j, k) \in V$  needs to learn  $A_{i,j}^{2^{t-1}}$ .<sup>12</sup> To do so,  $h(i, j, k)$  joins the multicast group of  $h(i, j)$  using [4, Theorem 2.3]. With the help of [4, Theorem 2.4],  $h(i, j)$  can then multicast  $A_{i,j}^{t-1}$  to all  $h(i, j, k)$ . Since there are  $L \leq [n_c]^3 = O(n)$  nodes  $h(i, j, k)$  that each join a multicast group, and each node needs to send and receive at most  $\ell = O(\log n)$  values, w.h.p., the theorems imply a runtime of  $O(\log n)$ , w.h.p.

After  $h(i, j, k)$  has received  $A_{i,j}^{2^{t-1}}$ , it sends it to both  $h(i, k, j)$  and  $h(j, k, i)$ . It is easy to see that thereby  $h(i, j, k)$  will receive  $A_{i,k}^{2^{t-1}}$  from  $h(i, k, j)$  and  $A_{k,j}^{2^{t-1}}$  from  $h(k, j, i)$ . Afterwards,  $h(i, j, k)$  sends the value  $A_{i,k}^{2^{t-1}} + A_{k,j}^{2^{t-1}}$  to  $h(i, j)$  by participating in an aggregation using [4, Theorem 2.2] and the minimum function, whereby  $h(i, j)$  receives  $A_{i,j}^{2^t}$ . By the same arguments as before,  $L = O(n)$ , and  $\ell = O(\log n)$ , which implies a runtime of  $O(\log n)$ , w.h.p.

► **Lemma 27.** After  $\lceil \log n \rceil + 2$  many matrix multiplications,  $h(i, j)$  stores  $d(i, j)$  for every  $i, j \in [n_c]$ . The total number of rounds is  $O(\log^2 n)$ , w.h.p.

## 7.4 Approximating SSSP and the Diameter

We are now all set in order to compute approximate distances between any two nodes  $s, t \in V$ . Specifically, we approximate  $d(s, t)$  by

$$\tilde{d}(s, t) = \min\{d_M(s, t), d(s, \sigma(s)) + d(\sigma(s), \sigma(t)) + d(\sigma(t), t)\}.$$

We now show that  $\tilde{d}(s, t)$  gives a 3-approximation for  $d(s, t)$ .

► **Lemma 28.** Let  $s, t \in V$  and  $d(s, t)$  be the length of the shortest path from  $s$  to  $t$ . It holds that  $d(s, t) \leq \tilde{d}(s, t) \leq 3d(s, t)$ .

To approximate SSSP, every node  $v$  needs to learn  $\tilde{d}(s, v)$  for a given source  $s$ . To do so, the nodes first have to compute  $d_M(s, v)$ , which can be done in time  $O(\log n)$  by performing SSSP in  $M$  using Theorem 5. Then, the nodes construct  $D_T$  in time  $O(\log^2 n)$  using Lemma 21. With the help of  $D_T$  and Lemma 24,  $s$  can compute  $d(s, \sigma(s))$ , which is then broadcast to all nodes in time  $O(\log n)$  using Lemma 3. Then, we compute all pairwise distances in  $G$  between all shortcut nodes in time  $O(\log^2 n)$ , w.h.p., using Lemma 27; specifically, every shortcut node  $v$  learns  $d(\sigma(s), v)$ . By performing a slight variant of the algorithm of Lemma 24, we can make sure that every node  $t$  not only learns its closest shortcut node  $\sigma(t)$  in  $M$ , but also retrieves  $d(\sigma(s), \sigma(t))$  from  $\sigma(t)$  within  $O(\log n)$  rounds. Since  $t$  is now able to compute  $\tilde{d}(s, t)$ , we conclude the following theorem.

<sup>12</sup> We will again ignore the fact that a node may have to act on behalf of at most  $O(\log n)$  nodes  $h(i, j, k)$ .



► **Theorem 29.** *3-approximate SSSP can be computed in graphs that contain at most  $n + O(n^{1/3})$  edges and have arboricity  $O(\log n)$  in time  $O(\log^2 n)$ , w.h.p.*

For a 3-approximation of the diameter, consider  $\tilde{D} = 2 \max_{s \in V} d(s, \sigma(s)) + \max_{x, y \in \Sigma} d(x, y)$ .  $\tilde{D}$  can easily be computed using Lemmas 21, 24, and 27, and by using Lemma 3 on  $M$  to determine the maxima of the obtained values. By the triangle inequality, we have that  $D \leq \tilde{D}$ . Furthermore, since  $d(s, \sigma(s)) \leq D$  and  $\max_{x, y \in \Sigma} d(x, y) \leq D$ , we have that  $\tilde{D} \leq 3D$ .

► **Theorem 30.** *A 3-approximation of the diameter can be computed in graphs that contain at most  $n + O(n^{1/3})$  edges and have arboricity  $O(\log n)$  in time  $O(\log^2 n)$ , w.h.p.*

---

## References

- 1 Dana Angluin, James Aspnes, Jiang Chen, Yinghua Wu, and Yitong Yin. Fast Construction of Overlay Networks. In *Proc. of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 145–154, 2005.
- 2 Mikhail Atallah and Uzi Vishkin. Finding euler tours in parallel. *J. of Computer and System Sciences*, 29(3):330–337, 1984.
- 3 John Augustine, Keerti Choudhary, Avi Cohen, David Peleg, Sumathi Sivasubramaniam, and Suman Sourav. Distributed graph realizations, 2020. [arXiv:2002.05376](https://arxiv.org/abs/2002.05376).
- 4 John Augustine, Mohsen Ghaffari, Robert Gmyr, Kristian Hinnenthal, Fabian Kuhn, Jason Li, and Christian Scheideler. Distributed computation in node-capacitated networks. In *Proc. of the 31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 69–79, 2019.
- 5 John Augustine, Kristian Hinnenthal, Fabian Kuhn, Christian Scheideler, and Philipp Schneider. Shortest paths in a hybrid network model. In *Proc. of the 2020 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1280–1299, 2020.
- 6 John Augustine and Sumathi Sivasubramaniam. Spartan: A framework for sparse robust addressable networks. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1060–1069, 2018.
- 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.
- 8 Surender Baswana and Sandeep Sen. A simple and linear time randomized algorithm for computing sparse spanners in weighted graphs. *Random Structures & Algorithms*, 30(4):532–563, 2007.
- 9 Boaz Ben-Moshe, Amit Dvir, Michael Segal, and Arie Tamir. Centdian computation in cactus graphs. *J. of Graph Algorithms and Applications*, 16(2):199–224, 2012.
- 10 Tao Chen, Xiaofeng Gao, and Guihai Chen. The features, hardware, and architectures of data center networks: A survey. *J. of Parallel and Distributed Computing*, 96:45–74, 2016.
- 11 Yong Cui, Hongyi Wang, and Xiuzhen Cheng. Channel allocation in wireless data center networks. In *Proc. of IEEE INFOCOM*, pages 1395–1403, 2011.
- 12 Hristo N. Djidjev, Grammati E. Pantziou, and Christos D. Zaroliagis. Computing shortest paths and distances in planar graphs. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 327–338, 1991.
- 13 Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yashaiah Fainman, George Papen, and Amin Vahdat. Helios: a hybrid electrical/optical switch architecture for modular data centers. In *Proc. of the ACM SIGCOMM 2010 conference*, pages 339–350, 2010.
- 14 Michael Feldmann, Kristian Hinnenthal, and Christian Scheideler. Fast hybrid network algorithms for shortest paths in sparse graphs, 2020. [arXiv:2007.01191](https://arxiv.org/abs/2007.01191).
- 15 Jie Gao and Li Zhang. Well-separated pair decomposition for the unit-disk graph metric and its applications. *SIAM J. Comput.*, 35(1):151–169, 2005.



- 16 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 Algorithms, Languages, and Programming (ICALP)*, pages 137:1–137:15, 2017.
- 17 Thorsten Götte, Kristian Hinnenthal, and Christian Scheideler. Faster construction of overlay networks. In *International Colloquium on Structural Information and Communication Complexity*, pages 262–276. Springer, 2019.
- 18 Thorsten Götte, Kristian Hinnenthal, Christian Scheideler, and Julian Werthmann. Time-optimal construction of overlay networks, 2020. [arXiv:2009.03987](#).
- 19 Joseph JaJa. *An Introduction to Parallel Algorithms*, volume 17. Addison Wesley, 1992.
- 20 Daniel Jung, Christina Kolb, Christian Scheideler, and Jannik Sundermeier. Competitive routing in hybrid communication networks. In *ALGOSENSORS*, volume 11410, pages 15–31, 2018.
- 21 Haim Kaplan, Wolfgang Mulzer, Liam Roditty, and Paul Seiferth. Routing in unit disk graphs. *Algorithmica*, 80(3):830–848, 2018.
- 22 Udit Narayana Kar and Debarshi Kumar Sanyal. An overview of device-to-device communication in cellular networks. *ICT Express*, 4(3):203–208, 2018.
- 23 Fabian Kuhn and Philipp Schneider. Computing shortest paths and diameter in the hybrid network model. In *2020 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 109–118, 2020.
- 24 Yu-Feng Lan and Yue-Li Wang. An optimal algorithm for solving the 1-median problem on weighted 4-cactus graphs. *European Journal of Operational Research*, 122(3):602–610, 2000.
- 25 Jason Li. Faster parallel algorithm for approximate shortest path. In *Proc. of the 52nd Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 308–321, 2020.
- 26 Danupon Nanongkai. Distributed approximation algorithms for weighted shortest paths. In *Proc. of the 46th Annual ACM Symposium on Theory of Computing (STOC)*, pages 565–573, 2014.
- 27 C. St. J. A. Nash-Williams. Decomposition of Finite Graphs Into Forests. *J. of the London Mathematical Society*, 39(1):12–12, 1964.
- 28 David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. Society for Industrial and Applied Mathematics, 2000.
- 29 Peter Robinson. Being fast means being chatty: The local information cost of graph spanners, 2020. [arXiv:2003.09895](#).
- 30 Michael Rosenberg and Guenter Schaefer. A survey on automatic configuration of virtual private networks. *Computer Networks*, 55(8):1684–1699, 2011.
- 31 Robert E. Tarjan and Uzi Vishkin. An Efficient Parallel Biconnectivity Algorithm. *SIAM J. on Computing*, 14(4):862–874, 1985.
- 32 Anis Tell, Wale Babalola, George Kalebiala, and Krishna Chinta. Sd-wan: A modern hybrid-wan to enable digital transformation for businesses. *IDC White Paper*, April 2018.
- 33 Jeffrey D. Ullman and Mihalis Yannakakis. High-probability parallel transitive-closure algorithms. *SIAM J. on Computing*, 20(1):100–125, 1991.

# Secured Distributed Algorithms Without Hardness Assumptions

Leonid Barenboim

Department of Mathematics and Computer Science, The Open University of Israel, Raanana, Israel  
leonidb@openu.ac.il

Harel Levin

Department of Physics, Nuclear Research Center-Negev, Giv'atayim, Israel  
Department of Mathematics and Computer Science, The Open University of Israel, Raanana, Israel  
harell@nrcn.org.il

---

## Abstract

---

We study algorithms in the distributed message-passing model that produce secured output, for an input graph  $G$ . Specifically, each vertex computes its part in the output, the entire output is correct, but each vertex cannot discover the output of other vertices, with a certain probability. This is motivated by high-performance processors that are embedded nowadays in a large variety of devices. Furthermore, sensor networks were established to monitor physical areas for scientific research, smart-cities control, and other purposes. In such situations, it no longer makes sense, and in many cases it is not feasible, to leave the whole processing task to a single computer or even a group of central computers. As the extensive research in the distributed algorithms field yielded efficient decentralized algorithms for many classic problems, the discussion about the security of distributed algorithms was somewhat neglected. Nevertheless, many protocols and algorithms were devised in the research area of secure multi-party computation problem (MPC or SMC). However, the notions and terminology of these protocols are quite different than in classic distributed algorithms. As a consequence, the focus in those protocols was to work for every function  $f$  at the expense of increasing the round complexity, or the necessity of several computational assumptions. In this work, we present a novel approach, which rather than turning existing algorithms into secure ones, identifies and develops those algorithms that are inherently secure (which means they do not require any further constructions). This approach yields efficient secure algorithms for various locality problems, such as coloring, network decomposition, forest decomposition, and a variety of additional labeling problems. Remarkably, our approach does not require any hardness assumption, but only a private randomness generator in each vertex. This is in contrast to previously known techniques in this setting that are based on public-key encryption schemes.

This paper is eligible for the best student paper award

**2012 ACM Subject Classification** Theory of computation → Distributed algorithms; Security and privacy → Privacy-preserving protocols; Mathematics of computing → Graph coloring; Mathematics of computing → Graph algorithms

**Keywords and phrases** distributed algorithms, privacy preserving, graph coloring, generic algorithms, multi-party computation

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2020.32

**Related Version** A full version of the paper is available at [5], <https://arxiv.org/abs/2011.07863>.

## 1 Introduction

Over the last few decades, computational devices get smaller and are embedded in a wide variety of products. High-performance processors are embedded in smart phones, wearable devices and smart home devices. Furthermore, sensor networks were established to monitor physical areas for scientific research, smart-cities control and other purposes. In such situations, it no longer makes sense, and in many cases it is not feasible, to leave the whole



© Leonid Barenboim and Harel Levin;

licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 32; pp. 32:1–32:16

Leibniz International Proceedings in Informatics



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

processing task to a single computer or even a group of central computers. In the distributed algorithms research field, all the processors are employed to solve a problem together. The basic assumption is that all the processors run the same program simultaneously. The network topology is represented by a graph  $G = (V, E)$  where each processor (also referred as *node*) is represented by a vertex,  $v \in V$ . Each communication line between a pair of processors  $v, u \in V$  in the network is represented by an edge  $(v, u) \in E$ .

The time complexity of a distributed algorithm is measured by rounds. Each round consists of three steps: (1) Each processor receives the messages that were sent by its neighbors on the previous round. (2) Each processor performs a local computation. (3) Each processor may send messages to its neighbors. The time complexity of distributed algorithms is measured by the number of rounds necessary to complete an algorithm. Local computations (that is, computations performed inside the nodes) are not taken into account in the running time analysis in this model.

Despite the extensive research in the distributed algorithms field in the last decades, the discussion about the security of distributed algorithms was somewhat neglected. Nevertheless, many protocols and algorithms were devised in the research area of cryptography and network security. The secure multi-party computation problem (MPC or SMC) is one of the main problems in the cryptography research. However, the notions and terminology of these protocols is quite different than in classic distributed algorithms. Moreover, most of these protocols assume the network forms a complete graph. Additionally, the protocols have no restriction on the amount of communication between the nodes.

In this work we devise secure distributed algorithms, in the sense that the output of each processor is not revealed to others, even though the overall solution expressed by all outputs is correct. Our notion of security is the following. Consider a problem where the goal is assigning a label to each vertex or edge of the graph  $G = (V, E)$ , out of a range  $[t]$ , for some positive  $t$ . A secure algorithm is required to compute a proper labeling, such that for any vertex  $v \in V$ , (respectively edge  $e \in E$ ) the other vertices in  $V$  (resp. edges in  $E$ ) are not aware of the label of  $v$  (resp.  $e$ ). Moreover, other vertices or edges can guess the label with probability at most  $1/\lambda$ , for an appropriate parameter  $\lambda \leq t$ . Note that this requirement can be achieved if each participant  $v$  (resp.  $e$ ) in the network computes a set of labels  $\{l^1, l^2, \dots, l^\lambda\}$  ( $l^i \in [t]$ ), such that any selection from its set forms a proper solution, no matter which selections are made in the sets of other participants. For example, in a proper coloring problem, if each vertex computes a set of colors (rather than just one color), and the set is disjoint from the sets of all its neighbors, the goal is achieved. In this case each participant draws a solution from its set of labels uniformly at random. The result is kept secret by the participant, and thus others can guess it with probability at most  $1/\lambda$ . Thus, if the number of labels is small, the possibility of guessing a result of a vertex becomes quite large, inevitably. As we will demonstrate later, one can artificially increase the amount of labels to achieve smaller probabilities. However, when it is impossible to use a large number of labels, other techniques can be taken into account (such as Parter and Yogev's compiler [20]). Nevertheless, our method is applicable to various distributed problems. Moreover, the overhead caused by the privacy preserving is negligible as the round complexity of our algorithms is similar to the best known (non privacy preserving) algorithms. A summary is found in Table 1. The parameter  $\lambda$  is referred to as the *solution domain* in Table 1. The ratio between  $t$  and  $\lambda$  is referred to as the *contingency factor*. These terms will be discussed later in Chapter 3.

■ **Table 1** List of inherently secure algorithms and their privacy attributes.

Problem	Type of Graph	Rounds Complexity	Solution Domain Size	Contingency Factor
$3\Delta$ -Coloring	Oriented trees	$O(\log^* n)$	$\Delta$	3
$2c \cdot \Delta \log n$ -Coloring	General	$O(1)$	$c \cdot \log n/2$	$O(\Delta)$
$O(\Delta^2)$ -Coloring	General	$\log^* n + O(1)$	$\Delta$	$O(\Delta)$
$p$ -Defective $O\left(\left(\frac{\Delta}{p}\right)^2\right)$ -Coloring	General	$O(\log^* n)$	$O\left(\frac{\Delta}{p}\right)$	$O\left(\frac{\Delta}{p}\right)$
$2a \cdot c \cdot \log n$ -Coloring	Bounded Arboricity $a$	$O(\log n)$	$O(\log n)/2$	$O(a)$
$(O(\log n), O(c \cdot \log n))$ -Network Decomposition	General	$O(\log^2 n)$	$c > 1$	$O(\log n)$
$\Delta$ -Forest Decomposition	General	$O(1)$	$\binom{\Delta}{ \Gamma(v) }$	1
$(2 + \epsilon) \cdot a$ -Forest Decomposition	Bounded Arboricity $a$	$O(\log n)$	$\binom{(2+\epsilon) \cdot a}{ \Gamma(v) }$	1
$O(\Delta \log n)$ -Edge Coloring	General	$O(1)$	$c \cdot \log n$	$O(\Delta)$
$O(\Delta^2)$ -Edge Coloring	General	$\log^* n + O(1)$	$(2\Delta - 1)$	$O(\Delta)$
$p$ -Defective $O\left(\left(\frac{\Delta}{p}\right)^2\right)$ -Edge Coloring	General	$O(1)$	$O\left(\left(\frac{\Delta}{p}\right)^2\right)$	1
$(t \cdot \sqrt{\Delta})$ -Edge Coloring of a Dominating Set	General	$\tilde{O}(\log \Delta + \log^3 \log n)$	$t$	$\sqrt{\Delta}$

## 2 Background

### 2.1 Distributed Algorithms

Given a network of  $n$  processors (or *nodes*), consider a graph  $G = (V, E)$  such that  $V = v_1, v_2, \dots, v_n$  is a set of vertices, each represents a processor. For each two vertices  $u, v \in V$ , there is an edge  $(u, v) \in E$  if and only if the two processors corresponding to the vertices  $u, v$  have a communication link between them. A communication link may be unidirectional or bidirectional, resulting in an undirected or a directed graph (respectively). Unless stated otherwise, the graphs in this work are simple, undirected and unweighted.

Two vertices  $u, v \in V$  are *independent* if and only if  $(u, v) \notin E$ . The *neighbors* set of a vertex  $v \in V$ ,  $\Gamma(v)$  consists of all the vertices in  $V$  that share a mutual edge with  $v$  in  $E$ . Formally,  $\Gamma(v) = \{u \in V | (u, v) \in E\}$ . The *degree* of a vertex  $v \in V$ ,  $deg(v) = |\Gamma(v)|$ . Note that  $0 \leq deg(v) \leq n - 1$ . The *maximum degree* of graph  $G$ ,  $\Delta(G)$ , is the degree of the vertex  $v \in V$  which has the maximum number of neighbors. If the graph  $G$  is directed, the out (respectively, in) degree of vertex  $v \in V$  ( $deg_{out}(v)$  and resp.  $deg_{in}(v)$ ) is the number of edges  $(u, v) \in E$  ( $u \in V$ ) with orientation that goes out from (respectively, in to) vertex  $v$ .

Throughout this paper,  $\mathcal{LOCAL}$  model will be used as the message-passing model. In this model, each communication line can send at each round an unrestricted amount of bits. It means that the primary measure is the number of rounds each node needs to “consult” its neighborhood by sending messages. This is in contrast to  $\mathcal{CONGEST}$  model, where the bandwidth on each communication line on each cycle is bounded by  $O(\log n)$ .

A single bit can pass from one endpoint of the graph to the other endpoint in  $D(G)$  rounds (where  $D(G)$  is the diameter of graph  $G$ ). Thus, in the  $\mathcal{LOCAL}$  model we usually look for time complexity lower than  $O(D(G))$  and even sub-logarithmic (in terms of  $|V|$ ), since all the nodes can learn the entire topology of the graph in  $O(D(G))$  rounds and then perform any computation on the entire graph. Consequently, the research in local distributed algorithms is focused on solving those graph theory problems which have solutions that depend on the local neighborhood of each vertex rather than the entire graph topology.

Most of the problems in graph theory may be classified into two types. The first type is a bipartition of the graph (whether the vertices, the edges, or both) into two sets. For some problems both of these sets are of interest, and for other problems only one of the sets, while the other sets may be categorized as “all the rest”. Examples of such problems include

Maximal Independent Set and Maximal Matching. The other type of problems partitions the graph into several sets. This type of problems may be referred as “labeling” problems where there is a set of valid labels and every part of the graph is labeled by a unique label. Examples of such problems include Coloring and Network Decomposition.

In the following sections we will present some of the problems in the field of graph theory which exploit the potential of distributed algorithms. While we discuss some of the main bipartition problems, in our model for privacy preserving the labeling problems are more relevant.

## 2.2 Graph Theory Problems

The definition of the problems is given here briefly, a detailed definition can be found in [5].

A function  $\varphi : V \rightarrow [\alpha]$  is a legal  $\alpha$ -Coloring of graph  $G = (V, E)$  if and only if, for each  $\{v, u\} \in E \rightarrow \varphi(v) \neq \varphi(u)$ . Similarly, a function  $\varphi : E \rightarrow [\alpha]$  is a valid  $\alpha$ -edge coloring of graph  $G = (V, E)$  i.f.f. for any vertex  $v \in V$  there are no two distinct vertices  $u, w \in \Gamma(v)$  such that  $\varphi((v, u)) = \varphi((v, w))$ . While a deterministic construction of such  $(\Delta + 1)$ -graph coloring requires at least  $O(\log^* n)$  rounds [16]. A randomized  $(\Delta + 1)$ -graph coloring can be done in  $\text{poly}(\log \log n)$  rounds [22]. Graph coloring is of special interest due to its applications in many resource management algorithms. In particular, certain resource allocation tasks require a proper coloring (possibly of a power graph) and that each vertex knows its own color, but not the colors of its neighbors. For example, this is the case in certain variants of Time Division Multiple Access schemes.

A *forest* is a graph which contains no cycles. A *forest decomposition* of graph  $G = (V, E)$  is an edge-disjoint partition of  $G$ , to  $\alpha$  sub-graphs  $\mathcal{F}_1, \mathcal{F}_2, \dots, \mathcal{F}_\alpha$  such that  $\mathcal{F}_i$  is a forest for every  $1 \leq i \leq \alpha$ . One way to define the *arboricity* of a graph  $G$  is as the minimal number of forests which are enough to fully cover  $G$ .

Given a graph  $G = (V, E)$  and a vertex-disjoint partition of graph  $G = (V, E)$  to  $\alpha$  clusters  $\mathcal{C}_1, \mathcal{C}_2, \dots, \mathcal{C}_\alpha$ , we define an auxiliary graph  $\mathcal{G} = (\mathcal{V}, \mathcal{E})$  such that  $\mathcal{V} = \{\mathcal{C}_1, \mathcal{C}_2, \dots, \mathcal{C}_\alpha\}$  and  $(\mathcal{C}_u, \mathcal{C}_v) \in \mathcal{E}$  ( $\mathcal{C}_u, \mathcal{C}_v \in \mathcal{V}$ ) iff  $\exists (u, v) \in E$  such that  $u \in \mathcal{C}_u$  and  $v \in \mathcal{C}_v$ . The partition  $\mathcal{C}_1, \mathcal{C}_2, \dots, \mathcal{C}_\alpha$  is a valid  $(d, c)$ -network decomposition [1] if (1) the chromatic number of  $\mathcal{G}$  is at most  $c$  and (2) the distance between each pair of vertices contained in the same cluster  $v, u \in \mathcal{C}_i$  is at most  $d$ . In strong network decomposition, the distance is measured with respect to the cluster  $\mathcal{C}_i$  (in other words,  $\text{dist}_{\mathcal{C}_i}(v, u) \leq d$ ). In weak network decomposition, the distance is measured with respect to the original graph  $G$  (in other words,  $\text{dist}_G(v, u) \leq d$ ). Different algorithms yield different kind of network decompositions which satisfy different values of  $d$  and  $c$ . One of the most valuable decompositions, which presents good trade-off between the radius of each cluster and the chromatic number of the auxiliary graph is an  $(O(\log n), O(\log n))$ -network decomposition.

A set  $I \subseteq V$  of vertices is called an *Independent Set (IS)* if and only if for each pair of vertices  $v, u \in I$  there is no edge  $(v, u) \in E$ . An independent set  $I$  is *Maximal Independent Set (MIS)* i.f.f. there is no vertex  $v \in V \setminus I$  such that  $I \cup \{v\}$  is a valid independent set. Similarly, a set of edges  $M \subseteq E$  is called a *Matching* i.f.f. there is no pair of vertices  $u_1, u_2 \in V$  ( $u_1 \neq u_2$ ) such that  $\exists v \in V$  where  $\{(u_1, v), (u_2, v)\} \subseteq M$ . A matching  $M$  is *Maximal Matching (MM)* i.f.f. there is no edge  $e \in E \setminus M$  such that  $M \cup \{e\}$  is a valid matching.

## 2.3 Secure Multi-Party Computation

*Multi-Party Computation (MPC)* is the ability of a party consisting of  $n$  participants to compute a certain function  $f(x_1, x_2, \dots, x_n)$  where each participant  $i$  ( $1 \leq i \leq n$ ) holds only its own input  $x_i$ . At the research fields of cryptography and networks security, *Secure MPC* [23] protocols enables parties to compute a certain function  $f$  without revealing their own input ( $x_i$ ). Our security model is information-theoretic secure, which means it is not based on any computational assumptions. Furthermore, we base our security notion on the *semi-honest model* (as was devised by [11]) which means that there are possibly curious participants but no *malicious adversary*. In other words, adversary participant can not deviate from the prescribed protocol. However, it may be curious, meaning it may run an additional computation in order to find out private data of another participants. Permitting the existence of malicious adversaries which may collude with  $t$  nodes will necessitate the graph to be  $(2t + 1)$ -connected for security to hold (as shown by [20]), which is not a feasible constraint.

Previous works on secure-MPC ([23], [11]) do not state any assumptions on the nature of neither the function  $f$  nor the interactions between the participants. As a consequence, the privacy preserving protocols devised during the past decades are generalized for any kind of mathematical function and not necessarily computation of graph features. Furthermore, each of the participants is assumed to be an equal part of the computation. As such, any pair of participants is assumed to have a private communication line of its own. Translating those protocols to distributed algorithms for graph theory problems, will require a complete graph representing the communication which may be different than the input graph of the problem. While this approach is applicable in many realistic networks and problems, general networks with non-uniform communication topology may benefit from efficient distributed algorithms for computations where the desired function  $f$  is local. Other works (such as [13] and [12]) are dedicated to general graphs. However, their goal was not to optimize the rounds complexity as the protocols created by their algorithms will require at least  $O(n^2)$  rounds even for a relatively simple function  $f$ . Furthermore, their techniques require a heavy setup phase, and based on some computational assumptions. Several other works provide secure protocols for general or sparse graphs ([6] [10] [7]). However, the focus in those protocols was to work for every function  $f$ , at the expense of increasing the round complexity, or the necessity of several computational assumptions.

Recently, Parter and Yogev [20] [21] suggested a new kind of privacy notion which they tailored to the *CONGEST* distributed model. In their notion, the neighbors of each node  $v$  construct a *private neighborhood tree* throughout which they broadcast a shared randomness. This randomness is used in order to encrypt the private variable of each neighbor. The node receives these encrypted private variables  $x_1, x_2, \dots, x_t$  ( $t = |\Gamma(v)|$ ) and performs its local computation  $f(x_1, x_2, \dots, x_t)$ . Let  $\text{OPT}(G)$  be the best depth possible for private neighborhood trees. Parter and Yogev devised an algorithm which constructs such trees in  $O(n + \Delta \cdot \text{OPT}(G))$  rounds, where each tree has depth of  $O(\text{OPT}(G) \cdot \text{polylog}(n))$  and each edge  $e \in E$  is part of at most  $O(\text{OPT}(G) \cdot \text{polylog}(n))$  trees. Using their notion one can turn any  $r$ -rounds algorithm into a secure algorithm with an overhead of  $\text{poly}(\Delta, \log n) \cdot \text{OPT}(G)$  rounds for each round. Furthermore, they showed that for a specific family of distributed algorithms (to which they referred as “simple”), the round overhead can be reduced to  $\text{OPT}(G) \cdot \text{polylog}(n)$ . Using their method they have devised a variety of both global and symmetry-breaking local algorithms. However, their notion requires an extensive pre-construction phase additional to the secure computation itself which requires quite high round complexity. Furthermore, their notion assumes a bridgeless graph, meaning that the

graph consists of a single connected component and there is no single edge such that its removal will split the graph into two connected components. The method of Parter and Yegorov relies on cryptographic hardness assumptions, e.g., the existence of a shared randomness and a public-key encryption scheme. This allows achieving security both for input and output. In contrast, our current work focuses on output security, but there are no hardness assumptions, and no requirement for bridgeless graphs.

### 3 Inherently Secure Distributed Algorithms

Most of the classic distributed algorithms models assume that each vertex is aware of its neighbors. This is the case in our paper as well. Usually each node does not have any additional input except for its own ID. The output of the algorithm is a set of labels where each label corresponds to each vertex. Throughout the current work, vertices IDs will not be considered as a private input. Formally, from the perspective of node  $v \in V$ , a classic distributed algorithm calculates a function  $f_v(D_{\Gamma(v)}) = l_v$ , where  $l_v \in [l]$  (for some constant  $l$ ) is vertex  $v$ 's label which was calculated based on the input messages ( $D_{\Gamma(v)}$ ) came from vertex  $v$ 's neighbors ( $\Gamma(v)$ ). From a global perspective, the algorithm computes:  $f : G(V, E) \rightarrow [l]^n$ . This work considers the following security notion: each vertex  $v$  cannot infer the value of  $l_u$  (such that  $u \in V, u \neq v$ ) with a certain probability. Our model assumes that each node  $v \in V$  holds a private randomness generator  $r_v$ .

As an example, consider an algorithm for  $\Delta^2$  graph multicoloring of graph  $G = (V, E)$  with maximum degree  $\Delta = \Delta(G)$  which provides any vertex  $v$  with a set of  $\Delta$  *valid* colors  $\varphi(v) = \{x_1, x_2, \dots, x_\Delta\}$ . By “valid” we mean that any of the colors in  $\varphi(v)$  is not contained in any of  $v$ 's neighbors' sets, i.e.  $x_i \notin \bigcup_{u \in \Gamma(v)} \varphi(u)$  (for any  $1 \leq i \leq \Delta$ ). Using this kind of coloring,  $v$  can *privately* select a random color out of the  $\Delta$  valid colors in  $\varphi(v)$ . Hence, the identity of the exact color of  $v$  can be securely hidden from any of the other vertices in  $G$ .

We generalize the above idea as follows. Consider the following family of algorithms. Each algorithm  $\Pi$  in the *Inherently-Secure* algorithms family  $\mathcal{IS}$  consists of two stages: (1) Calculating a generic set of  $k$  possible valid labels. (2) Randomly and privately (using the private randomness generator  $r_v$ ), each node selects its final label. That is, the first stage of algorithm  $\Pi$  (denoted by  $\Pi_{generic}$ ) calculates the function:  $f_1(G(V, E)) = \{\ell_{u_1}, \dots, \ell_{u_n}\}$ , where  $\ell_{u_i} = \{l_i^1, l_i^2, \dots, l_i^k\}$  for any  $u_i \in V$ . Henceforth,  $\Pi_{generic}$  will be referred as *generic-algorithm*. The first stage can run without any additional security considerations, meaning any node may know the  $\ell_{u_i}$  of other nodes. Later, we will show algorithms which satisfies even stronger security notion where the identity of  $\ell_{u_i}$  is also kept secret. The second stage ( $\Pi_{select}$ ) securely calculates the function  $f_2 : [l]^k \rightarrow [l]^n$ . Overall, algorithm  $\Pi$  indeed calculates  $f = f_1 \circ f_2 : G(V, E) \rightarrow \{l_1, \dots, l_n\}$ . Let  $L$  be the ground set of valid labels from which the possible labels are being picked, i.e. for any  $1 \leq i \leq n$  and  $1 \leq j \leq k$ ,  $l_i \in L$  and  $l_i^j \in L$ .

By increasing the amount of possible values ( $k$ ) we make the actual labels  $\{l_1, l_2, \dots, l_n\}$  less predictable. However, in order to do so we may need to increase the ground set of the available labels. For instance, in graph coloring we may want to be able to produce  $\Delta$  valid possible colors for each vertex. However, an increase of the amount of colors (to  $\Delta^2$ ) may be necessary. On the other hand, one may want to minimize the size of the ground set since large ground sets may lead to trivial algorithms on one hand, and to a higher memory complexity on the other hand.

In order to analyze this kind of algorithms we define several parameters.



► **Definition 1.** *The size of the problem domain of a problem  $\mathcal{P}$  solved by algorithm  $\Pi$  which calculates the function  $f : G(V, E) \rightarrow \{l_1, l_2, \dots, l_n\}$ , where  $l_i \in L$ , is the number of valid labels for any  $l_i$ , i.e.  $|L|$ .*

► **Definition 2.** *The size of the solution domain of a generic algorithm  $\Pi_{generic}$  which calculates the function  $f_1 : G(V, E) \rightarrow \{\{l_1^1, l_1^2, \dots, l_1^k\}, \dots, \{l_n^1, l_n^2, \dots, l_n^k\}\}$  is the minimal number of valid possible labels for any vertex, i.e.  $k$ .*

► **Definition 3.** *The contingency factor of generic algorithm  $\Pi_{generic}$  used to solve problem  $\mathcal{P}$  (as they defined on definitions 1 and 2) is the ratio between the size of the problem domain  $|L|$  (Def. 1) and the size of the solution domain  $k$  (Def. 2), i.e.  $|L|/k$ .*

In order to clarify these definitions, consider the problem of  $\Delta^2$ -graph coloring. The size of the problem domain is  $\Delta^2$ . A generic algorithm that calculates  $\Delta$  possible valid colors for each vertex will provide a solution domain of size  $\Delta$ . The contingency factor of this algorithm will be  $\frac{\Delta^2}{\Delta} = \Delta$ .

In many cases, the number of labels (i.e. the size of the problem domain) can be increased artificially by a factor  $c > 1$ . This artificial increase will lead to an expansion of the problem domain by the same factor  $c$ . As a result, the contingency factor will remain the same. That is, the contingency factor is a property of the algorithm itself and not influenced by artificial increases. Small contingency factor indicates that most of the members of the problem domain are valid options on the solution domain, while the generic algorithm did not exclude those members from being considered as valid possible solutions. As such, small contingency factor indicates that the algorithm preserves better security by excluding only a small portion of possible solutions. Problems with small problem-domain will have even smaller solution domain which will lead to a contingency factor that is close to the original size of the problem domain. Therefore, finding a generic algorithm with good contingency factor for these problems is a complicated task. As a consequence, we will focus on finding generic algorithms for problems with relatively large problem-domain, i.e. labeling problems. For problems with small problem domain, other techniques (such as Parter and Yogev's compiler [20]) should be considered.

Note that even though a malicious node may interrupt the validity of the algorithm by picking a solution which is not part of its solution domain, this kind of intrusion will not affect the privacy of the algorithm. However, in our model the nodes are not malicious.

### 3.1 Generic Algorithms for Graph-Coloring

Considering the problem of graph coloring, we will focus on finding generic algorithms that will provide contingency factor of  $\Delta$ . This factor is optimal for general graphs, as we prove in Theorem 4.

► **Theorem 4.** *For any  $\alpha$ -coloring problem ( $\alpha > \Delta$ ), and any generic algorithm  $\Pi$ , there is an infinite family of graphs such that their solution domain must be of size  $O(\alpha/\Delta)$  at most. Hence, the contingency factor would be at least  $\Omega(\Delta)$ .*

**Proof.** Suppose for contradiction that there is a valid solution domain such that every vertex has more than  $\alpha/\Delta$  valid options. Consider a graph  $G = (V, E)$  with clique  $C \subseteq V$  of size  $|C| = \Delta + 1$ . Each vertex  $v \in C$  has  $\Delta$  neighbors, each of them has  $\alpha/\Delta$  valid colors. But since  $v$  and all its neighbors are part of the clique, each of them has a unique set of colors. It means that there are at least  $(\Delta + 1) \cdot (\alpha/\Delta) > \alpha$  colors in the  $\alpha$ -coloring, which is a contradiction. ◀

### 3.1.1 Generic Algorithm for $3\Delta$ -Coloring of Oriented Trees

A well known algorithm for 3-Coloring of oriented trees was devised by Cole and Vishkin [8]. The deterministic algorithm exploits the asymmetric relationship between a vertex  $v$  and its parent (in the tree)  $\pi(v)$  in order to get a valid coloring in  $O(\log^* n)$  rounds.

For any two integers  $a$  and  $b$ , let  $\langle a, b \rangle$  be a tuple which can be represented in binary as the concatenation of the binary representations of  $a$  and  $b$ . We can define a generic algorithm that runs Cole Vishkin's algorithm to get a valid coloring  $\varphi : V \rightarrow [3]$ . Later, each vertex will set its solution domain  $\hat{\varphi}(v)$  as follows:  $\hat{\varphi}(v) = \bigcup_{0 \leq i < \Delta} \langle i, \varphi(v) \rangle$ . As a result, we get a generic algorithm where each vertex has  $\Delta$  valid colors. The validity of this algorithm is provided by the following Lemma:

► **Lemma 5.** *For any vertex  $v \in V$ , for every value  $x \in \hat{\varphi}(v)$ ,  $x$  is not a possible color for any other vertex  $u \in \Gamma(v)$ .*

**Proof.** Suppose for contradiction that there exists a vertex  $u \in \Gamma(v)$  such that  $x \in \hat{\varphi}(u)$ . Since,  $\hat{\varphi}(v) = \bigcup_{0 \leq i < \Delta} \langle i, \varphi(v) \rangle$ , there exists a value  $1 \leq i \leq \Delta$  such that  $x = \langle i, \varphi(v) \rangle = \langle i, \varphi(u) \rangle$ . Hence,  $\varphi(v) = \varphi(u)$ , which is a contradiction since  $\varphi$  is a valid coloring as was proved by [8]. ◀

Lemma 5 leads to the following corollary:

► **Corollary 6.** *Given a tree  $T = (V, E, v)$ , there exists a generic algorithm which provides any vertex  $v \in V$  with a set of  $\Delta$  possible valid colors*

The generic algorithm described above uses a simple approach which achieves privacy by artificially increasing the size of both the problem domain and the solution domain accordingly. Another technique is to run an algorithm  $d$  times in parallel. For coloring problems a good  $d$  will probably be  $\Delta$  (as was shown in Theorem 4). While this approach will lead in deterministic algorithms to the same results as the previous technique, applying this technique with random algorithms will lead to solution domain which is somewhat less predictable than the domain we will receive by artificially increasing the size of the solution domain.

While these approaches (artificially increasing the size of the solution domain and run the algorithm multiple times) are useful for problems with very efficient base algorithms (i.e. Cole Vishkin 3-coloring), for many problems such an efficient algorithm is not yet known. However, one still may devise efficient generic-algorithms for some of these problems, as demonstrated in the following sections.

### 3.1.2 Generic Algorithm for $2\Delta c \cdot \log n$ -Coloring of General Graphs

While the best known algorithms for  $(\Delta + 1)$ -Coloring of generic graphs uses logarithmic number of rounds [14] [18], a reasonable size of contingency factor may yield more efficient algorithms with sub-logarithmic and even constant number of rounds. As an example, consider Algorithm 1 which uses  $O(1)$  rounds to achieve a secure coloring of general graphs with contingency factor of  $O(\Delta)$ .

■ **Algorithm 1** GENERIC-RANDOM-COLORING.

---

**Result:** A set of  $O(\log n)$  colors for each vertex  $v \in V$

- 1 Every vertex selects independently at random  $k = c \cdot \log n$  different numbers ( $c$  is a constant,  $c > 1$ )  $I = \{\langle 1, x_1 \rangle, \langle 2, x_2 \rangle, \dots, \langle k, x_k \rangle\}$  where  $x_i \in [2\Delta]$  is a number selected uniformly at random (for each  $1 \leq i \leq k$ ).
  - 2 Send  $I$  to each neighbor.
  - 3 For each message  $\hat{I} = \{\langle 1, \hat{x}_1 \rangle, \langle 2, \hat{x}_2 \rangle, \dots, \langle k, \hat{x}_k \rangle\}$  received, **do**  $I \leftarrow I \setminus \hat{I}$ .
-

The fact that Algorithm 1 is privacy preserving is established by the Theorem below. Its proof can be found in [5]. The contingency factor of the algorithm is  $\frac{2\Delta c \cdot \log n}{c \cdot \log n/2} = O(\Delta)$ .

► **Theorem 7.** *For any vertex  $v \in V$ , executing algorithm *GENERIC-RANDOM-COLORING*, the algorithm produces a set of at least  $k/2$  valid colors in  $O(1)$  rounds, with high probability.*

### 3.1.3 Generic Algorithm for $O(\Delta^2)$ -Coloring of General Graphs

Since currently known deterministic algorithms for  $(\Delta + 1)$ -coloring require at least  $\sqrt{\log n}$  rounds, applying the simultaneous execution described above with such an algorithm will lead to relatively poor round complexity. Instead, in this section we generalize a construction of [3][16] which provides  $O(\Delta^2)$ -coloring in  $\log^* n + O(1)$  rounds, in order to directly (i.e. without simultaneous executions) obtain secure algorithm for  $O(\Delta^2)$ -coloring. We employ a Lemma due to Erdős et al. [9].

► **Lemma 8.** *For two integers  $n$  and  $\Delta$ ,  $n > \Delta \geq 4$ , there exists a family  $\mathcal{J}$  of  $n$  subsets of the set  $\{1, \dots, m\}$ ,  $m = \lceil \Delta^2 \cdot \ln n \rceil$ , such that if  $F_0, F_1, \dots, F_\Delta \in \mathcal{J}$  then  $F_0 \not\subseteq \bigcup_{i=1}^{\Delta} F_i$ .*

A set system  $\mathcal{J}$  which satisfies the above is referred as  $\Delta$ -cover-free set.

Erdős et al. [9] also showed an algebraic construction which satisfies Lemma 8. For two integers  $n$  and  $\Delta$ , using a ground set of size  $m = O(\Delta^2 \cdot \log^2 n)$ , they construct a family  $\mathcal{F}$  of  $n$  subsets of the set  $\{1, \dots, m\}$ , such that  $\mathcal{F}$  is a  $\Delta$ -cover-free. Linial [16] showed that this construction can be utilized for distributed graph coloring. We construct a slightly different family which also provides multiple uncovered elements in each set:

► **Theorem 9.** *For two integers  $n$  and  $\Delta$ , using a ground set of size  $m = O(\Delta^2 \cdot \log^2 n)$ , there exists a family  $\mathcal{F}$  of  $n$  subsets of the set  $\{1, \dots, m\}$ , such that if  $F_0, F_1, \dots, F_\Delta \in \mathcal{F}$  then  $\left| F_0 \setminus \bigcup_{i=1}^{\Delta} F_i \right| \geq \Delta$ .*

The proof of Theorem 9 can be found in [5].

These polynomials provides sets of labels such that if every vertex is assigned to a set, each set has at least  $\Delta$  values which are not contained in any of its neighbors' sets. An illustration of this construction is provided in [5].

Next, we will use the constructions from [9] and Theorem 9 to devise a generic-algorithm for  $O(\Delta^2)$ -coloring. Our algorithm is similar to Linial's iterative algorithm ([16]), but instead of getting only one color on the last iteration, we get  $\Delta$  different possible colors (for each vertex).

Starting with a valid  $n$ -coloring for some graph  $G = (V, E)$  (the color of each vertex is its ID), we can apply the coloring algorithm from [16] which will turn the  $n$ -coloring into an  $O(\Delta^2 \log^2 n)$ -coloring in a single round. After  $\log^* n + O(1)$  rounds we will get an  $O(\Delta^2 \log^2 \Delta)$ -coloring. For a sufficiently large  $\Delta$ , it holds that  $O(\Delta^2 \log^2 \Delta) \leq (3\Delta)^3$ . Hence, in order to further reduce the number of the colors to  $O(\Delta^3)$  and get  $\Delta$  valid optional colors we will use the set system from Theorem 9 to reduce the  $O(\Delta^2 \log^2 \Delta)$ -coloring to a  $q^2 = (3\Delta)^2$ -coloring of  $G$  such that each vertex has at least  $\Delta$  valid colors. To conclude, the problem domain is of size  $9\Delta^2$ . The solution domain contains of at least  $\Delta$  valid colors. Consequently, the contingency factor is  $O(\Delta)$ , which is proved to be optimal (see Theorem 4).

### 3.1.4 Generic Algorithm for $p$ -Defective $O\left(\left(\frac{\Delta}{p}\right)^2\right)$ -Coloring of General Graphs

For a graph  $G = (V, E)$ , the function  $\varphi : V \rightarrow [\alpha]$  is a valid  $p$ -defective  $\alpha$ -coloring iff for each vertex  $v \in V$ , the number of neighbors which have the same color as  $v$  is at most  $p$ , i.e.  $|\{u \in \Gamma(v) \mid \varphi(v) = \varphi(u)\}| \leq p$ .

First, we shall find the lower bound of contingency factors for generic algorithms of defective coloring:

► **Theorem 10.** *For any  $p$ -defective  $\alpha$ -coloring problem, there is an infinite family of graphs such that their solution domain must be of size  $O\left(\alpha \cdot \frac{p}{\Delta}\right)$  at most and the contingency factor will be at least  $\Omega\left(\frac{\Delta}{p}\right)$ .*

**Proof.** Suppose for contradiction that there is a valid solution domain such that every vertex has more than  $\alpha \cdot \frac{p}{\Delta}$  valid options. Consider a graph  $G = (V, E)$  with a clique  $C \subseteq V$  of size  $|C| = \Delta + 1$ . Each vertex  $v \in C$  has  $\Delta$  neighbors, each of them has  $\alpha \cdot \frac{p}{\Delta}$ . Since  $v$  and all its neighbors are part of a clique, each color can be an optional color of at most  $p$  different vertices. It means that there are at least  $(\Delta + 1) \cdot (\alpha \cdot \frac{p}{\Delta}) / p > \alpha$  colors in the  $p$ -defective  $\alpha$ -coloring, which is a contradiction. ◀

The results from the previous section can be extended and combined with the results of [4], to achieve generic algorithm for  $\rho$ -defective  $O\left(\left(\frac{\Delta}{\rho}\right)^2\right)$ -coloring which provides a solution domain of size  $O\left(\frac{\Delta}{\rho}\right)$ . Such an algorithm provides an optimal contingency factor. The proof of the next Theorem can be found in [5].

► **Theorem 11.** *Given a graph  $G = (V, E)$  ( $|V| = n$ ) with maximum degree  $\Delta$ , and a fixed parameter  $1 \leq p \leq \Delta$ , there is a generic algorithm that calculates  $p$ -defective  $O\left(\left(\frac{\Delta}{p}\right)^2\right)$ -coloring with a solution domain of size  $O(\Delta/p)$  and a contingency factor of at least  $\Omega(\Delta/p)$ , in  $O(\log^* n)$  rounds.*

The contingency factor is optimal by Theorem 10.

## 3.2 Generic Algorithm for Network Decomposition

As was mentioned before, network decomposition may be referred as a labeling problem where the cluster IDs are the labels and the clusters assignment is the labeling function. Hence, network decomposition problems are good candidates for generic algorithms. However, considering the term of privacy in the network decomposition problem, different definitions may be suggested. One may suggest a permissive notion where all the members of the same cluster are allowed to share their private data with each other. This permissive notion makes sense since network decomposition is frequently used as a building block in other algorithms (such as coloring or finding MIS) where in the first stage each vertex discovers its cluster's topology, calculates private solution for the entire cluster, and then communicates with other clusters in order to generate an overall solution. However, even on a restrictive notion where each vertex may know only its own cluster assignment, some efficient algorithms may be suggested. Hence, during this work we will use the restrictive notion.

In section 3.1.1 we have shown how multiple simultaneous executions of the same random algorithm can expand the solutions domain and provide a generic algorithm for graph coloring problems. This approach can be adopted in order to expand the solution domain

of network decomposition algorithms, However, since the network decomposition should satisfy certain constraints (namely, the depth of the clusters and the chromatic number of the auxiliary graph), this approach should be implemented carefully. We will use the weak-diameter  $(O(\log n), O(\log n))$ -network decomposition random algorithm devised by Linial and Saks [17]. This algorithm runs in  $O(\log^2 n)$  rounds. Our generic algorithm proceeds as follows. Given a graph  $G = (V, E)$ , and a positive integer  $c > 1$ , execute Linial and Saks's algorithm for  $c$  times simultaneously, in parallel. Each of the execution will have its own serial number  $i \in \{1, \dots, c\}$ . Let  $C_i : V \rightarrow \{1, \dots, O(\log n)\}$  be a set of labeling functions ( $1 \leq i \leq c$ ), such that  $C_i(v) = j$  iff vertex  $v \in V$  was assigned by the  $i$ -th execution to cluster  $C_j$ . For each vertex  $v \in V$  we assign a set of  $\log n$  different possible labels  $C(v) = \langle 1, C_1(v) \rangle, \dots, \langle c, C_c(v) \rangle$ . Each of the labels will represent a distinct cluster ID. These labels are different since even if the independent executions produced the same cluster assignments, the first parameter on each tuple representing the label will be different since it represents the unique ID of each independent execution.

The clusters assignment described above is a privacy preserving  $(O(\log n), O(c \cdot \log n))$ -network decomposition. The proof of the following Theorem can be found in [5]

► **Theorem 12.** *Given a graph  $G = (V, E)$ , there is a generic algorithm which calculates weak-diameter  $(O(\log n), O(c \cdot \log n))$ -network decomposition in  $O(\log^2 n)$  rounds. The algorithm produces  $c$  valid possible cluster assignments for each vertex  $v \in V$ .*

Since the size of the problem domain is  $O(c \cdot \log n)$  and the solution domain is of size  $c$ , the contingency factor is  $O(\log n)$ .

Usually, it is useful to set the parameters of the network decomposition to be polylogarithmic in  $n$ . Hence, it may be useful to set  $c = \log n$  and get an  $(O(\log n), O(\log^2 n))$ -network decomposition. On the other hand, in order to preserve privacy, setting  $c = \min(\Delta, \log n)$  is sufficient as it allows each of the  $\Delta$  neighbors of each vertex to have a different set of possible cluster assignments.

### 3.3 Generic Algorithms for Forest Decomposition

An *oriented tree* is a directed tree  $T = (V, E, r)$  where  $r \in V$  is the root vertex, where every vertex  $v \in V$  knows the identity of its parent  $\pi(v)$  and has an oriented edge  $(v, \pi(v))$ . An *oriented forest* is such a graph that any of its connected components are oriented trees. Any graph  $G = (V, E)$  with maximum degree  $\Delta$  can be decomposed into a set of  $\Delta$  edge-disjoint forests  $F_1, \dots, F_\Delta (F_i = (V_{F_i}, E_{F_i}))$  such that  $E = \bigcup_{1 \leq i \leq \Delta} E_{F_i}$ . The problem of how to decompose a graph into forests can be viewed as a labeling problem where each edge should have a label  $1 \leq i \leq \Delta$  that represents the forest  $F_i$  which it belongs to. Since in every oriented forest, each vertex has at most 1 parent, each vertex will have at most  $\Delta$  outgoing edges, each belongs to a different forest. Hence, for each vertex  $v \in V$  there are  $\binom{\Delta}{\deg_{out}(v)}$  different options to associate edges to different forests. From the edge's point of view, each of the  $\Delta$  labels is a valid possible label.

Panconesi and Rizzi [19] devised an algorithm for  $\Delta$ -forest decomposition of a general undirected graph in 2 rounds. Their algorithm can be viewed as two separate algorithms, each of a single round. The first algorithm is a simple yet powerful way to decompose a directed acyclic graph with maximum outgoing degree  $d$  into  $d$  oriented forests. The second is a way to turn an undirected graph with maximum degree  $\Delta$  into a directed acyclic graph with maximum outgoing degree  $\Delta$ . Each of these algorithms run in a single round. Combining these two algorithms produces a 2 round algorithm for  $\Delta$ -forest decomposition of any undirected graph with maximum degree  $\Delta$ .

Next we will describe Panconesi and Rizzi's algorithm for forest decomposition of oriented graphs. We will show how this algorithm can be modified in order to preserve privacy while maintaining a contingency factor of 1. Later, we will show two algorithms which produce a directed acyclic graphs. The first is Panconesi and Rizzi's algorithm for orienting any general undirected graph. The second algorithm (due to [2]) performs an acyclic orientation for a graph with bounded arboricity  $a$  such that the maximum outgoing degree is  $\lfloor 2 + \epsilon \rfloor \cdot a$ . The combination of these algorithms yields a  $\Delta$ -forest decomposition for graphs with maximum degree  $\Delta$  and  $\lfloor 2 + \epsilon \rfloor \cdot a$ -forest decomposition for graphs with bounded arboricity  $a$ . Both of them fit the constraint of preserving privacy.

### 3.3.1 Forest Decomposition of Oriented Graphs

Given a directed acyclic graph  $G = (V, E)$ , such that each vertex has a set of outgoing edges  $E(v) = \{(v, u) \mid (v, u) \in E\}$  the single round algorithm of Panconesi and Rizzi [19] goes as follows. Each vertex  $v \in V$ , in parallel, assigns a distinct number  $1 \leq i \leq |E(v)|$  to each  $e \in E(v)$ . Let  $\hat{E}_i$  be the set of all edges that were assigned with the number  $i$ . The forest decomposition is the set of forests  $F_1, \dots, F_\Delta$  where  $F_i = (V, \hat{E}_i)$ . The correctness of the algorithm was proved by [19]. While in the original algorithm the nodes do not assign the labels randomly, in our algorithm a random assignment is required. Next, we analyze the privacy of the algorithm. The proof of the following Theorem can be found in [5].

► **Theorem 13.** *Panconesi and Rizzi's forest decomposition algorithm with random label assignment is privacy preserving and it has a contingency factor of 1.*

### 3.3.2 Acyclic Orientation of Graphs

Panconesi and Rizzi [19] showed that the simple orientation where each edge is oriented towards the vertex with the higher ID, is an acyclic orientation. Hence, any undirected graph can achieve an acyclic orientation in a single round, and can be decomposed privately into  $\Delta$  forests in one additional round. This orientation provides each vertex with up to  $\Delta!$  valid options for forest assignments. From the edge's point of view, each of the  $\Delta$  labels is a valid possible label. Barenboim et al. [2] devised an  $O(\log n)$ -rounds algorithm that receives a graph with bounded arboricity  $a$  and performs an acyclic orientation with maximum outgoing degree of  $\lfloor 2 + \epsilon \rfloor \cdot a$ . This orientation is achieved by partitioning the vertices of a graph  $G$  into  $l = \lfloor \frac{2}{\epsilon} \log n \rfloor$  sets  $H_1, \dots, H_l$  such that each vertex  $v \in H_i$  ( $i \in \{1, \dots, l\}$ ) has at most  $(2 + \epsilon) \cdot a$  neighbors in  $\cup_{j=i}^l H_j$ . Then, the orientation is done such that each edge  $(u, v) \in E$  with endpoints  $u \in H_i$  and  $v \in H_j$ , points towards the vertex that belongs to the higher ranked set (in case the two endpoints belong to the same set, the edge will point towards the vertex with the higher ID). This orientation provides each vertex with up to  $\binom{\lfloor 2 + \epsilon \rfloor \cdot a}{|E(v)|}$  valid options for forest assignments. From the edge's point of view, each of the  $\lfloor 2 + \epsilon \rfloor \cdot a$  labels is a valid possible label.

## 3.4 Generic Algorithms for Graph Coloring of Graphs With Bounded Arboricity $a$

The forest decomposition algorithms that was described above can be used as building blocks for other distributed algorithms for classic graph theory problems as graph coloring. In the following chapter we will use the  $\lfloor 2 + \epsilon \rfloor \cdot a$ -forest decomposition of [2] that we showed in the previous chapter to achieve an  $2a \cdot c \cdot \log n$ -coloring for graphs with bounded arboricity  $a$  (for any  $c > 1$ ). We will show that this coloring is private and has contingency factor of  $O(a)$ .



Combining the GENERIC-RANDOM-COLORING algorithm (algorithm 1) with the acyclic orientation algorithm devised by [2] yields a secure algorithm for generic  $2a \cdot c \cdot \log n$ -Coloring for graphs with bounded arboricity  $a$  such that from initial selection of  $k = c \cdot \log n$  initial colors (for any  $c > 1$ ), each vertex has, at the end of the execution, at least  $k/2$  valid optional colors. The algorithm basically performs the original random generic coloring, but it makes advantage of the acyclic orientation to break the symmetry between each pair of neighbors and make sure that only  $O(a)$  neighbors constraint the valid residual colors of each vertex.

The algorithm consists of two steps. On the first step, the algorithm performs an acyclic orientation of graph  $G$  such that the maximum outgoing degree is  $\lfloor 2 + \epsilon \rfloor \cdot a$ . The orientation is done by invoking the first two steps of procedure Forests-Decomposition (algorithm 2 in [2]) with graph  $G$  and parameter  $0 < \epsilon \leq 2$ . On the second step the generic coloring is done. Each vertex  $v$  chooses independently at random  $k = c \cdot \log n$  numbers from the range  $[2 \cdot A]$ . These choices form a set of optional colors:  $I_v = \{ \langle 1, x_1 \rangle, \dots, \langle k, x_k \rangle \}$ . Next, each vertex  $v$  sends its set of colors  $I_v$  to its children (in correspondence to the orientation). Each vertex  $u$  which received a set  $I_v$  from one of its parents performs  $I_u \leftarrow I_u \setminus I_v$ .

► **Lemma 14.** *The residual set of colors contains at least  $k/2$  colors.*

The proof of this Lemma can be found in [5].

► **Lemma 15.** *For any vertex  $v \in V$ , there is no color  $\langle i, x_i \rangle$  in the residual available colors set  $I_v$  such that  $\langle i, x_i \rangle \in \bigcup_{u \in \Gamma(v)} I_u$ .*

**Proof.** Suppose for contradiction that  $\langle i, x_i \rangle \in I_u$  for some  $u \in \Gamma(v)$ . Let  $F_j$  be the forest in  $\mathcal{F}$  which includes the edge  $(u, v)$ . It means that either  $u \in \pi_j(v)$  or  $v \in \pi_j(u)$ , which means that either  $u$  or  $v$  received it from its parent ( $v$  or  $u$ , respectively) and should have removed it from its residual set, contradiction. ◀

The time complexity of the algorithm follows from the time complexity of Procedure Forest-Decomposition( $a, \epsilon$ ), which is  $O(\log n)$ , plus  $O(1)$  for coloring. The size of the problem domain is  $2a \cdot c \cdot \log n$  and the size of the solution domain is  $\frac{c \cdot \log n}{2}$ . Hence, the contingency factor is  $O(a)$ .

### 3.5 Generic Algorithms for Edge Coloring

When considering the meaning of privacy in the context of edges, there is a slight difference between vertex coloring and edge coloring. Since the algorithms in both *LOCAL* and *CONGEST* ran on the vertices (rather than the edges, which represents communication lines) the color of each vertex should be known only to the vertex itself. On the other hand, in edge coloring, both edge endpoints are responsible for the coloring of the edge, which means that in terms of privacy preserving we may consider the edge coloring as private when at most the two endpoints of each edge know the color of the edge. However, when the graph is directed, we may demand that only the source endpoint of the edge will be aware of edge's color.

Nevertheless, there is a strong connection between graph vertex coloring to edge coloring. The similarity between the two problems is obvious, but more interestingly, there is a straight reduction between vertex coloring and edge coloring algorithms for general graphs. In the following section we will use this reduction in order to perform privacy preserving generic edge coloring of graphs. This reduction can be used to apply the defective graph coloring we



presented in section 3.1.4 in order to compute a defective edge coloring. There is however, a faster way to get a defective edge coloring. This technique, which is due to [15] will be presented in the later section.

### 3.5.1 Edge Coloring Using Line Graphs

Given a graph  $G = (V, E)$ , a *line graph*  $L(G)$  of graph  $G$  is a graph which is constructed as follows. Each edge  $e \in E$  becomes a vertex of the line graph  $L(G) = (E, \mathcal{E})$ . Each two distinct vertices of the line graph  $e_1, e_2 \in E$  are connected  $((e_1, e_2) \in \mathcal{E})$  if they are incident to a single vertex in the original graph, i.e. there exist three vertices  $v, u, w \in V$  such that  $e_1 = (v, u)$  and  $e_2 = (v, w)$ . Observe that the line graph has  $m \leq n^2$  vertices and a maximum degree of  $\Delta(L(G)) = (2\Delta(G) - 1)$ . Also observe that a legal vertex coloring in the line graph  $L(G)$  is a legal edge coloring in the original graph. Hence, if any vertex of the original graph is responsible for the coloring of part of its incident edges, the vertices can produce a legal graph coloring for the line graph and translate it to a legal edge coloring of the original graph. The assignment of each vertex to any incident edge can be done by specifying that for any edge  $(u, v) \in E$ , the vertex with the greater ID is responsible for the coloring of the edge in the line graph.

As a result, the algorithms provided in section 3.1 can be applied to the line graph in order to produce a generic edge coloring. Given a graph  $G = (V, E)$  with maximum degree  $\Delta$ , the algorithm for  $2\Delta c \cdot \log n$ -Coloring, applied on the line graph  $L(G)$ , produces an  $2 \cdot (2\Delta - 1) \cdot c \cdot \log(n^2) = 8\Delta \cdot c \cdot \log n$ -edge-coloring, with solution domain of size  $c \cdot \log(n^2)/2 = c \cdot \log n$ , which yields a contingency factor of  $O(\Delta)$ . The algorithm for  $O(\Delta^2)$ -Coloring, applied on the line graph  $L(G)$ , produces an  $O((2\Delta - 1)^2) = O(\Delta^2)$  edge coloring of the original graph, with a solution domain of size  $(2\Delta - 1)$ , which keeps a contingency factor of  $O(\Delta)$ .

### 3.5.2 Generic Defective Edge Coloring

The line graph, which was presented in the previous section, can be used in order to perform a  $p$ -defective  $O\left(\left(\frac{2\Delta-1}{p}\right)^2\right)$ -generic edge coloring of the original graph by applying the algorithm from Theorem 11 on the line graph. Such an algorithm will achieve a solution domain of size  $O(\Delta/p)$  and a contingency factor of  $O(\Delta/p)$  as well, both in  $O(\log^* n)$  rounds. There is however a faster privacy preserving algorithm for  $p$ -defective  $O\left(\left(\frac{2\Delta-1}{p}\right)^2\right)$ -defective coloring based on the algorithm of Kuhn (Algorithm 3 in [15]).

Kuhn's algorithm goes as follows. Suppose we have an undirected graph  $G = (V, E)$  with maximum degree  $\Delta$ , and a constant  $i \geq 1$ . Each vertex numbers its adjacent edges with numbers between  $\{1, \dots, \lceil \Delta/i \rceil\}$  such that each number will be assigned to at most  $i$  of the vertex's adjacent edges. Then each vertex sends the number of each of its adjacent edges to the vertex on the other endpoint of the edge. Suppose that for a graph  $G = (V, E)$ , and an edge  $(u, v) \in E$ ,  $e_u$  and  $e_v$  are the colors that was assigned to edge  $e$  by vertex  $u$  and  $v$  (respectively). The set  $\{e_u, e_v\}$  is assigned to be the color of edge  $e$ . Kuhn showed that this simple  $O(1)$  rounds algorithm achieves a  $4i - 2$ -defective  $\binom{\lceil \Delta/i \rceil + 1}{2}$ -Edge Coloring. Setting  $p = 4i - 2$  we get an  $p$ -defective  $O\left(\left(\frac{\Delta}{p}\right)^2\right)$ -edge coloring.

Kuhn's algorithm performs communication only between the two endpoints of each edge and only once. Hence the knowledge about the color of each edge is held only by its two endpoints. While in Kuhn's algorithm the nodes do not assign the labels randomly, in our

algorithm a random assignment is required. Therefore, our algorithm preserves privacy. Since every vertex assigns the numbers independently, each of the available colors may be assigned (in certain scenario) to each edge. As a result, we achieve the following theorem.

► **Theorem 16.** *There is a privacy preserving algorithm for  $p$ -defective  $O\left(\left(\frac{\Delta}{p}\right)^2\right)$ -edge coloring with contingency factor of 1.*

### 3.6 Generic Algorithm for Edge Dominating Set Colored with $O(\sqrt{\Delta})$ Colors

The problem of constructing an edge domination set, is a bipartition. However, a useful variant of this problem, where the dominating set should be colored with  $O(\sqrt{\Delta})$  colors is actually a labeling problem. A privacy preserving algorithm for this problem can be found in [5].

## 4 Conclusion

The computer-science research fields of secure multi-party computation and distributed algorithms were both highly investigated during the last decades. While both of the fields prosper and yield many theoretical and practical results, the connection between these fields was made only seldom. Nevertheless, as implementation of distributed algorithms becomes common in sensor networks and IoT (Internet of Things) architectures, efficient privacy preserving techniques are essential.

In this work we present a novel approach, which rather than turning existing algorithms into secure ones, identifies and develops those algorithms that are inherently secure. Naturally, our work focuses on labeling problems. The inherently secure algorithms analyzed in this work are listed in

We believe that these results establishes a broad basis for further research of both inherently secure algorithm and efficient techniques to translate distributed algorithms into secure algorithms. Such algorithms will open new possibilities for secure interconnection between machines, eliminating the need to mediate through a central secure server. As a consequence, distributed communication would possibly open a free and secure way to transmit data and solve problems.

---

## References

- 1 Baruch Awerbuch, Michael Luby, Andrew V Goldberg, and Serge A Plotkin. Network decomposition and locality in distributed computation. In *30th Annual Symposium on Foundations of Computer Science*, pages 364–369. IEEE, 1989.
- 2 Leonid Barenboim and Michael Elkin. Sublogarithmic distributed mis algorithm for sparse graphs using nash-williams decomposition. *Distributed Computing*, 22(5-6):363–379, 2010.
- 3 Leonid Barenboim and Michael Elkin. Distributed graph coloring: Fundamentals and recent developments. *Synthesis Lectures on Distributed Computing Theory*, 4(1):1–171, 2013.
- 4 Leonid Barenboim, Michael Elkin, and Fabian Kuhn. Distributed  $(\delta+1)$ -coloring in linear (in  $\delta$ ) time. *SIAM Journal on Computing*, 43(1):72–95, 2014.
- 5 Leonid Barenboim and Harel Levin. Secured distributed algorithms without hardness assumptions, 2020. [arXiv:2011.07863](https://arxiv.org/abs/2011.07863).
- 6 Elette Boyle, Shafi Goldwasser, and Stefano Tessaro. Communication locality in secure multi-party computation. In *Theory of Cryptography Conference*, pages 356–376. Springer, 2013.

- 7 Nishanth Chandran, Juan Garay, and Rafail Ostrovsky. Improved fault tolerance and secure computation on sparse networks. In *International Colloquium on Automata, Languages, and Programming*, pages 249–260. Springer, 2010.
- 8 Richard Cole and Uzi Vishkin. Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 206–219. ACM, 1986.
- 9 Paul Erdős, Peter Frankl, and Zoltán Füredi. Families of finite sets in which no set is covered by the union of others. *Israel Journal of Mathematics*, 51(1):79–89, 1985.
- 10 Juan A Garay and Rafail Ostrovsky. Almost-everywhere secure computation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 307–323. Springer, 2008.
- 11 Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *Proc. of the 19th Symp. on Theory of Computing*, pages 218–229, 1987.
- 12 Shai Halevi, Yuval Ishai, Abhishek Jain, Ilan Komargodski, Amit Sahai, and Eylon Yogev. Non-interactive multiparty computation without correlated randomness. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 181–211. Springer, 2017.
- 13 Shai Halevi, Yuval Ishai, Abhishek Jain, Eyal Kushilevitz, and Tal Rabin. Secure multiparty computation with general interaction patterns. In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science*, pages 157–168. ACM, 2016.
- 14 Öjvind Johansson. Simple distributed  $\delta+1$ -coloring of graphs. *Information Processing Letters*, 70(5):229–232, 1999.
- 15 Fabian Kuhn. Weak graph colorings: distributed algorithms and applications. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 138–144. ACM, 2009.
- 16 Nathan Linial. Distributive graph algorithms global solutions from local data. In *Foundations of Computer Science, 1987.*, pages 331–335. IEEE, 1987.
- 17 Nathan Linial and Michael Saks. Low diameter graph decompositions. *Combinatorica*, 13(4):441–454, 1993.
- 18 Michael Luby. Removing randomness in parallel computation without a processor penalty. *Journal of Computer and System Sciences*, 47(2):250–286, 1993.
- 19 Alessandro Panconesi and Romeo Rizzi. Some simple distributed algorithms for sparse networks. *Distributed computing*, 14(2):97–100, 2001.
- 20 Merav Parter and Eylon Yogev. Distributed computing made secure: A new cycle cover theorem. *arXiv preprint arXiv:1712.01139*, 2017.
- 21 Merav Parter and Eylon Yogev. Secure distributed computing made (nearly) optimal. In *Proc. of 38th Symp. on Principles of Distributed Computing*, pages 107–116, 2019.
- 22 Václav Rozhoň and Mohsen Ghaffari. Polylogarithmic-time deterministic network decomposition and distributed derandomization. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, pages 350–363, 2020.
- 23 Andrew C Yao. Protocols for secure computations. In *Foundations of Computer Science, 1982. SFCS'08.*, pages 160–164. IEEE, 1982.

# Uniform Bipartition in the Population Protocol Model with Arbitrary Communication Graphs

**Hiroto Yasumi**

Nara Institute of Science and Technology, Japan  
yasumi.hiroto.yf9@is.naist.jp

**Fukuhito Ooshita**

Nara Institute of Science and Technology, Japan  
f-oosita@is.naist.jp

**Michiko Inoue**

Nara Institute of Science and Technology, Japan  
kounoe@is.naist.jp

**Sébastien Tixeuil**

Sorbonne Université, CNRS, LIP6, Paris, France  
Sebastien.Tixeuil@lip6.fr

---

## Abstract

In this paper, we focus on the uniform bipartition problem in the population protocol model. This problem aims to divide a population into two groups of equal size. In particular, we consider the problem in the context of *arbitrary* communication graphs. As a result, we investigate the solvability of the uniform bipartition problem with arbitrary communication graphs when agents in the population have designated initial states, under various assumptions such as the existence of a base station, symmetry of the protocol, and fairness of the execution. When the problem is solvable, we present protocols for uniform bipartition. When global fairness is assumed, the space complexity of our solutions is tight.

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

**Keywords and phrases** population protocol, uniform bipartition, distributed protocol

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2020.33

**Related Version** A full version of the paper is available at [34], <https://arxiv.org/abs/2011.08366>.

**Funding** This work was supported by JSPS KAKENHI Grant Numbers 18K11167, 20H04140, 20J21849, and JST SICORP Grant Number JPMJSC1806. This work was partially funded by the ANR project SAPPORO, ref. 2019-CE25-0005-1.

## 1 Introduction

### 1.1 Background

In this paper, we consider the population protocol model introduced by Angluin et al. [5]. The population protocol model is an abstract model for low-performance devices. In the population protocol model, devices are represented as anonymous agents, and a population is represented as a set of agents. Those agents move passively (*i.e.*, they cannot control their movements), and when two agents approach, they are able to communicate and update their states (this pairwise communication is called an interaction). A computation then consists of an infinite sequence of interactions.



© Hiroto Yasumi, Fukuhito Ooshita, Michiko Inoue, and Sébastien Tixeuil;  
licensed under Creative Commons License CC-BY

24th International Conference on Principles of Distributed Systems (OPODIS 2020).

Editors: Quentin Bramas, Rotem Oshman, and Paolo Romano; Article No. 33; pp. 33:1–33:16



Leibniz International Proceedings in Informatics

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

Application domains for population protocols include sensor networks used to monitor live animals (each sensor is attached to a small animal and monitors *e.g.* its body temperature) that move unpredictably (hence, each sensor must handle passive mobility patterns). Another application domain is that of molecular robot networks [28]. In such systems, a large number of molecular robots collectively work inside a human body to achieve goals such as transport of medicine. Since those robots are tiny, their movement is uncontrollable, and robots may only maintain extremely small memory.

In the population protocol model, many researchers have studied various fundamental problems such as leader election protocols [4] (A population protocol solves leader election if starting from an initially uniform population of agents, eventually a single agent outputs *leader*, while all others output *non-leader*), counting [8, 10, 11] (The counting problem consists in counting how many agents participate to the protocol; As the agents' memory is typically constant, this number is output by a special agent that may maintain logarithmic size memory, the base station), majority [6] (The majority problem aims to decide which, if any, initial state in a population is a majority),  $k$ -partition [32, 35, 36] (The  $k$ -partition problem consists in dividing a population into  $k$  groups of equal size), etc.

In this paper, we focus on the uniform bipartition problem [33, 35, 36], whose goal is to divide a population into two stable groups of equal size (the difference is one if the population size is odd). To guarantee the stability of the group, each agent eventually belongs to a single group and never changes the group after that. Applications of the uniform bipartition include saving batteries in a sensor network by switching on only one group, or executing two tasks simultaneously by assigning one task to each group. Contrary to previous work that considered *complete* communication graphs [33, 36], we consider the uniform bipartition problem over *arbitrary* graphs. In the population protocol model, most existing works consider the complete communication graph model (every pairwise interaction is feasible). However, realistic networks command studying incomplete communication graphs (where only a subset of pairwise interactions remains feasible) as low-performance devices and unpredictable movements may not yield a complete set of interactions. Moreover, in this paper, we assume the designated initial states (i.e., all agents share the same given initial state), and consider the problem under various assumptions such as the existence of a base station, symmetry of the protocol, and fairness of the execution. Although protocols with arbitrary initial states tolerate a transient fault, protocols with designated initial states can usually be designed using fewer states, and exhibit faster convergence times. Actually, it was shown in [35] that, with arbitrary initial states, constant-space protocols cannot be constructed in most cases even assuming complete graphs.

## 1.2 Related Works

The population protocol model was proposed by Angluin et al. [5], who were recently awarded the 2020 Edsger W. Dijkstra prize in Distributed Computing for their work. While the core of the initial study was dedicated to the computability of the model, subsequent works considered various problems (*e.g.*, leader election, counting, majority, uniform  $k$ -partition) under different assumptions (*e.g.*, existence of a base station, fairness, symmetry of protocols, and initial states of agents).

The *leader election* problem was studied from the perspective of time and space efficiency. Doty and Soloveichik [20] proved that  $\Omega(n)$  expected parallel time is required to solve leader election with probability 1 if agents have a constant number of states. Relaxing the number of states to a polylogarithmic value, Alistarh and Gelashvili [3] proposed a leader election protocol in polylogarithmic expected stabilization time. Then, Gaşieniec et al. [23] designed

■ **Table 1** The minimum number of states to solve the uniform bipartition problem with designated initial states over *complete* graphs [33, 35].

base station	fairness	symmetry	upper bound	lower bound
initialized/non-initialized base station	global	asymmetric	3	3
		symmetric	3	3
	weak	asymmetric	3	3
		symmetric	3	3
no base station	global	asymmetric	3	3
		symmetric	4	4
	weak	asymmetric	3	3
		symmetric	unsolvable	

a protocol with  $O(\log \log n)$  states and  $O(\log n \cdot \log \log n)$  expected time. Furthermore, the protocol of Gaşieniec et al. [23] is space-optimal for solving the problem in polylogarithmic time. In [29], Sudo et al. presented a leader election protocol with  $O(\log n)$  states and  $O(\log n)$  expected time. This protocol is time-optimal for solving the problem. Finally, Berenbrink et al. [15] proposed a time and space optimal protocol that solves the leader election problem with  $O(\log \log n)$  states and  $O(\log n)$  expected time. In the case of arbitrary communication graphs, it turns out that self-stabilizing leader election is impossible [7] (a protocol is self-stabilizing if its correctness does not depend on its initial global state). This impossibility can be avoided if oracles are available [9, 18] or if the self-stabilization requirement is relaxed: Sudo et al. [30] proposed a loosely stabilizing protocol for leader election (loose stabilization relates to the fact that correctness is only guaranteed for a very long expected amount of time).

The *counting* problem was introduced by Beauquier et al. [11] and popularized the concept of a base station. Space complexity was further reduced by follow-up works [10, 24], until Aspnes et al. [8] finally proposed a time and space optimal protocol. On the other hand, by allowing the initialization of agents, the counting protocols without the base station were proposed for both exact counting [16] and approximate counting [1, 16]. In [1], Alistarh et al. proposed a protocol that computes an integer  $k$  such that  $\frac{1}{2} \log n < k < 9 \log n$  in  $O(\log n)$  time with high probability using  $O(\log n)$  states. After that, Berenbrink et al. [16] designed a protocol that outputs either  $\lfloor \log n \rfloor$  or  $\lceil \log n \rceil$  in  $O(\log^2 n)$  time with high probability using  $O(\log n \cdot \log \log n)$  states. Moreover, in [16], they proposed the exact counting protocol that computes  $n$  in  $O(\log n)$  time using  $\tilde{O}(n)$  states with high probability.

The *majority* problem was addressed under different assumptions (*e.g.*, with or without failures [6], deterministic [22, 25] or probabilistic [2, 12, 13, 25] solutions, with arbitrary communication graphs [27], etc.). Those works also consider minimizing the time and space complexity. Berenbrink et al. [14] show trade-offs between time and space for the problem.

To our knowledge, the *uniform  $k$ -partition* problem and its variants have only been considered in complete communication graphs. Lamani et al. [26] studied a group decomposition problem that aims to divide a population into groups of designated sizes. Yasumi et al. [32] proposed a uniform  $k$ -partition protocol with no base station. Umino et al. [31] extended the result to the  $R$ -generalized partition problem that aims at dividing a population into  $k$  groups whose sizes follow a given ratio  $R$ . Also, Delporte-Gallet et al. [19] proposed a  $k$ -partition protocol with relaxed uniformity constraints: the population is divided into  $k$  groups such that in any group, at least  $n/(2k)$  agents exist, where  $n$  is the number of agents.

Most related to our work is the uniform bipartition solution for complete communication graphs provided by Yasumi et al. [33, 35]. For the uniform bipartition problem over complete graphs with designated initial states, Yasumi et al. [33, 35] studied space complexity under

■ **Table 2** The minimum number of states to solve the uniform bipartition problem with designated initial states over *arbitrary* graphs.  $P$  is a known upper bound of the number of agents, and  $l \geq 3$  and  $h$  are positive integers.

base station	fairness	symmetry	upper bound	lower bound
initialized/non-initialized base station	global	asymmetric	$3^*$	$3^\dagger$
		symmetric	$3^*$	$3^\dagger$
	weak	asymmetric	$3P + 1^*$ $3l + 1$ for no $l \cdot h$ cycle *	$3^\dagger$
		symmetric	$3P + 1^*$ $3l + 1$ for no $l \cdot h$ cycle *	$3^\dagger$
no base station	global	asymmetric	$4^*$	$4^*$
		symmetric	$5^*$	$5^*$
	weak	asymmetric	unsolvable*	
		symmetric	unsolvable <sup>†</sup>	

\* Contributions of this paper

† Deduced from Yasumi et al. [35]

various assumptions such as: (i) an initialized base station, a non-initialized base station, or no base station (an initialized base station has a designated initial state, while a non-initialized has an arbitrary initial state), (ii) asymmetric or symmetric protocols (asymmetric protocols allow interactions between two agents with the same state to map to two resulting different states, while symmetric protocols do not allow such a behavior), and (iii) global or weak fairness (weak fairness guarantees that every individual pairwise interaction occurs infinitely often, while global fairness guarantees that every recurrently reachable configuration is eventually reached). Furthermore, they also study the solvability of the uniform bipartition problem with arbitrary initial states. Table 1 shows the minimum number of states to solve the uniform bipartition with designated initial states over complete communication graphs.

There exist some protocol transformers that transform protocols for some assumptions into ones for other assumptions. In [5], Angluin et al. proposed a transformer that transforms a protocol with complete communication graphs into a protocol with arbitrary communication graphs. This transformer requires the quadruple state space and works under global fairness. In this transformer, agents exchange their states even after convergence. For the uniform bipartition problem, since agents must keep their groups after convergence, they cannot exchange their states among different groups and thus the transformer proposed in [5] cannot directly apply to the uniform bipartition problem. Bournez et al. [17] proposed a transformer that transforms an asymmetric protocol into symmetric protocol by assuming additional states. In [17], only protocols with complete communication graphs were considered and the transformer works under global fairness. We use the same idea to construct a symmetric uniform bipartition protocol under global fairness without a base station.

### 1.3 Our Contributions

In this paper, we study the solvability of the uniform bipartition problem with designated initial states over arbitrary graphs. A summary of our results is presented in Table 2. Let us first observe that, as complete communication graphs are a special case of arbitrary communication graphs, the impossibility results by Yasumi et al. [35] remain valid in our setting. With a base station (be it initialized or non-initialized) under global fairness, we extend the three states protocol by Yasumi et al. [35] from complete communication graphs to arbitrary communication graphs. With a non-initialized base station under weak fairness,



we propose a new symmetric protocol with  $3P + 1$  states, where  $P$  is a known upper bound of the number of agents. These results yield identical upper bounds for the easier cases of asymmetric protocols and/or initialized base station. In addition, we also show a condition of communication graphs in which the number of states in the protocol can be reduced from  $3P + 1$  to constant. Concretely, we show that the number of states in the protocol can be reduced to  $3l + 1$  if we assume communication graphs such that every cycle either includes the base station or its length is not a multiple of  $l$ , where  $l$  is a positive integer at least three. On the other hand, with no base station under global fairness, we prove that four and five states are necessary and sufficient to solve uniform bipartition with asymmetric and symmetric protocols, respectively. In the same setting, in complete graphs, three and four states were necessary and sufficient. So, one additional state enables problem solvability in arbitrary communication graphs in this setting. With no base station under weak fairness, we prove that the problem cannot be solved, using a similar argument as in the impossibility result for leader election by Fischer and Jiang [21]. Overall, we show the solvability of uniform bipartition in a variety of settings for a population of agents with designated initial states assuming arbitrary communication graphs. In cases where the problem remains feasible, we provide upper and lower bounds with respect to the number of states each agent maintains, and in all cases where global fairness can be assumed, our bounds are tight.

In this paper, because of space limitations, we omitted proofs of lemmas and theorems (see the full version [34]).

## 2 Definitions

### 2.1 Population Protocol Model

A population whose communication graph is arbitrary is represented by an undirected connected graph  $G = (V, E)$ , where  $V$  is a set of agents, and  $E \subseteq V \times V$  is a set of edges that represent the possibility of an interaction between two agents. That is, two agents  $u \in V$  and  $v \in V$  can interact only if  $(u, v) \in E$  holds. A protocol  $\mathcal{P} = (Q, \delta)$  consists of  $Q$  and  $\delta$ , where  $Q$  is a set of possible states for agents, and  $\delta$  is a set of transitions from  $Q \times Q$  to  $Q \times Q$ . Each transition in  $\delta$  is denoted by  $(p, q) \rightarrow (p', q')$ , which means that, when an interaction between an agent  $x$  in state  $p$  and an agent  $y$  in state  $q$  occurs, their states become  $p'$  and  $q'$ , respectively. Moreover, we say  $x$  is an initiator and  $y$  is a responder. When  $x$  and  $y$  interact as an initiator and a responder, respectively, we simply say that  $x$  interacts with  $y$ . 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  $\mathcal{P} = (Q, \delta)$  is symmetric if, for every transition  $(p, q) \rightarrow (p', q')$  in  $\delta$ ,  $(q, p) \rightarrow (q', p')$  exists in  $\delta$ . In particular, if a protocol  $\mathcal{P} = (Q, \delta)$  is symmetric and transition  $(p, p) \rightarrow (p', q')$  exists in  $\delta$ ,  $p' = q'$  holds. If a protocol is not symmetric, the protocol is asymmetric. Protocol  $\mathcal{P} = (Q, \delta)$  is deterministic if, for any pair of states  $(p, q) \in Q \times Q$ , exactly one transition  $(p, q) \rightarrow (p', q')$  exists in  $\delta$ . We consider only deterministic protocols in this paper. A global state of a population is called a configuration, defined as a vector of (local) 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 between two configurations  $C$  and  $C'$  is described as  $C \rightarrow C'$ , and means that configuration  $C'$  is obtained from  $C$  by a single interaction between two agents. 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 infinite sequence of configurations  $\Xi = C_0, C_1, C_2, \dots$  is an execution of a protocol if  $C_i \rightarrow C_{i+1}$  holds for

every  $i$  ( $i \geq 0$ ). An execution  $\Xi$  is weakly-fair if, for each pair of agents  $(v, v') \in E$ ,  $v$  (resp.  $v'$ ) interacts with  $v'$  (resp.,  $v$ ) infinitely often<sup>1</sup>. An execution  $\Xi$  is globally-fair if, for every 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 every possible interaction in  $C$  also occurs infinitely often. Then, if  $C$  occurs infinitely often,  $C'$  satisfying  $C \rightarrow C'$  occurs infinitely often, 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 the base station: initialized base station, non-initialized base station, and no base station. In the model with a base station, we assume that a single agent, called a base station, exists in  $V$ . Then,  $V$  can be partitioned into  $V_b$ , the singleton set containing the base station, and  $V_p$ , the set of agents except for the base station. The base station can be distinguished from other agents in  $V_p$ , although agents in  $V_p$  cannot be distinguished. Then, the state set  $Q$  can be partitioned into a state set  $Q_b$  for the base station, and a state set  $Q_p$  for agents in  $V_p$ . The base station has unlimited resources (with respect to the number of states), in contrast with other resource-limited agents (that are allowed only a limited number of states). So, when we evaluate the space complexity of a protocol, we focus on the number of states  $|Q_p|$  for agents in  $V_p$  and do not consider the number of states  $|Q_b|$  that are allocated to the base station. In the sequel, we thus say a protocol uses  $x$  states if  $|Q_p| = x$  holds. When we assume an initialized base station, the base station has a designated initial state. When we assume a non-initialized base station, the base station has an arbitrary initial state (in  $Q_b$ ), although agents in  $V_p$  have the same designated initial state. When we assume no base station, there exists no base station and thus  $V = V_p$  holds. For simplicity, we use agents only to refer to agents in  $V_p$  in the following sections. To refer to the base station, we always use the term base station (not an agent). In the initial configuration, both the base station and the agents are not aware of the number of agents, yet they are given an upper bound  $P$  of the number of agents. However, in protocols except for a protocol in Section 3.2, we assume that they are not given  $P$ .

## 2.2 Uniform Bipartition Problem

Let  $f : Q_p \rightarrow \{\text{red}, \text{blue}\}$  be a function that maps a state of an agent to *red* or *blue*. We define the color of an agent  $a$  as  $f(s(a))$ . Then, we say that agent  $a$  is *red* (resp., *blue*) if  $f(s(a)) = \text{red}$  (resp.,  $f(s(a)) = \text{blue}$ ) holds. If an agent  $a$  has state  $s$  such that  $f(s) = \text{red}$  (resp.,  $f(s) = \text{blue}$ ), we call  $a$  a *red* agent (resp., a *blue* agent). For some population  $V$ , the number of *red* agents (resp., *blue* agents) in  $V$  is denoted by  $\#\text{red}(V)$  (resp.,  $\#\text{blue}(V)$ ). When  $V$  is clear from the context, we simply write  $\#\text{red}$  and  $\#\text{blue}$ .

A configuration  $C$  is stable with respect to the uniform bipartition if there exists a partition  $\{H_r, H_b\}$  of  $V_p$  that satisfies the following conditions:

1.  $||H_r| - |H_b|| \leq 1$  holds, and
2. For every configuration  $C'$  such that  $C \xrightarrow{*} C'$ , each agent in  $H_r$  (resp.,  $H_b$ ) remains *red* (resp., *blue*) in  $C'$ .

---

<sup>1</sup> We use this definition for the lower bound under weak fairness, but for the upper bound we use a slightly weaker version. We show that our proposed protocols for weak fairness works if, for each pair of agents  $(v, v') \in E$ ,  $v$  and  $v'$  interact infinitely often (i.e., for interactions by some pair of agents  $v$  and  $v'$ , it is possible that  $v$  only becomes an initiator and  $v'$  never becomes an initiator).

An execution  $\Xi = C_0, C_1, C_2, \dots$  solves the uniform bipartition problem if  $\Xi$  includes a configuration  $C_t$  that is stable for uniform bipartition. Finally, a protocol  $\mathcal{P}$  solves the uniform bipartition problem if every possible execution  $\Xi$  of protocol  $\mathcal{P}$  solves the uniform bipartition problem.

### 3 Upper Bounds with a Non-initialized Base Station

In this section, we prove some upper bounds on the number of states that are required to solve the uniform bipartition problem over arbitrary graphs with designated initial states and a non-initialized base station. More concretely, with global fairness, we propose a symmetric protocol with three states by extending the protocol by Yasumi et al. [35] from a complete communication graph to an arbitrary communication graph. In the case of weak fairness, we present a symmetric protocol with  $3P + 1$  states, where  $P$  is a known upper bound of the number of agents.

#### 3.1 Upper Bound for Symmetric Protocols under Global Fairness

The state set of agents in this protocol is  $Q_p = \{initial, red, blue\}$ , and we assume that  $f(initial) = f(red) = red$  and  $f(blue) = blue$  hold. The designated initial state of agents is *initial*. The idea of the protocol is as follows: the base station assigns *red* and *blue* to agents whose state is *initial* alternately. As the base station cannot meet every agent (the communication graph is arbitrary), the positions of state *initial* are moved throughout the communication graph using transitions. Thus, if an agent with *initial* state exists somewhere in the network, the base station has infinitely many chances to interact with a neighboring agent with *initial* state. This implies that the base station is able to repeatedly assign *red* and *blue* to neighboring agents with *initial* state unless no agent anywhere in the network has *initial* state. Since the base station assigns *red* and *blue* alternately, the uniform bipartition is completed after no agent has *initial* state.

To make *red* and *blue* alternately, the base station has a state set  $Q_b = \{b_{red}, b_{blue}\}$ . Using its current state, the base station decides which color to use for the next interaction with a neighboring agent with *initial* state. Now, to move the position of an *initial* state in the communication graph, if an agent with *initial* state and an agent with *red* (or *blue*) state interact, they exchange their states. This implies that eventually an agent adjacent to the base station has *initial* state and then the agent and the base station interact (global fairness guarantees that such interaction eventually happens). Transition rules of the protocol are the following (for each transition rule  $(p, q) \rightarrow (p', q')$ , transition rule  $(q, p) \rightarrow (q', p')$  exists, but we omit the description).

1.  $(b_{red}, initial) \rightarrow (b_{blue}, red)$
2.  $(b_{blue}, initial) \rightarrow (b_{red}, blue)$
3.  $(blue, initial) \rightarrow (initial, blue)$
4.  $(red, initial) \rightarrow (initial, red)$

From these transition rules, the protocol converges when no agent has *initial* state (indeed, no interaction is defined when no agent has *initial* state).

► **Theorem 1.** *In the population protocol model with a non-initialized base station, there exists a symmetric protocol with three states per agent that solves the uniform bipartition problem with designated initial states assuming global fairness in arbitrary communication graphs.*

■ **Algorithm 1** Uniform bipartition protocol with  $3P + 1$  states.

---

**Variables at the base station:**

$RB \in \{r, b\}$ : The state that the base station assigns next

**Variables at an agent  $x$ :**

$color_x \in \{ini, r, b\}$ : Color of the agent, initialized to  $ini$

$depth_x \in \{\perp, 1, 2, 3, \dots, P\}$ : Depth of agent  $x$  in a tree rooted at the base station, initialized to  $\perp$

```

1: when an agent  $x$  and the base station interact do
2:   if  $color_x = ini$  and  $depth_x = 1$  then
3:      $color_x \leftarrow RB$ 
4:      $RB \leftarrow \overline{RB}$ 
5:   if  $depth_x = \perp$  then  $depth_x \leftarrow 1$ 
6: when two agents  $x$  and  $y$  interact do
7:   if  $depth_y \neq \perp$  and  $depth_x = \perp$  then  $depth_x \leftarrow depth_y + 1$ 
8:   else if  $depth_x \neq \perp$  and  $depth_y = \perp$  then  $depth_y \leftarrow depth_x + 1$ 
9:   if  $depth_x < depth_y$  and  $color_y = ini$  then
10:     $color_y \leftarrow color_x$ 
11:     $color_x \leftarrow ini$ 
12:   if  $depth_y < depth_x$  and  $color_x = ini$  then
13:     $color_x \leftarrow color_y$ 
14:     $color_y \leftarrow ini$ 

```

**Note:** If  $depth_x = \perp$  holds,  $color_x = ini$  holds.

---

Note that, under weak fairness, this protocol does not solve the uniform bipartition problem. This is because we can construct a weakly-fair execution of this protocol such that some agents keep *initial* state infinitely often. For example, we can make an agent keep *initial* by constructing an execution in the following way.

- If the agent (in *initial*) interacts with an agent in *red* or *blue*, the next interaction occurs between the same pair of agents.

## 3.2 Upper Bound for Symmetric Protocols under Weak Fairness

### 3.2.1 A protocol over arbitrary graphs

In this protocol, every agent  $x$  has variables  $color_x$  and  $depth_x$ . Variable  $color_x$  represents the color of agent  $x$ . That is, for an agent  $x$ , if  $color_x = ini$  or  $color_x = r$  holds,  $f(s(x)) = red$  holds. On the other hand, if  $color_x = b$  holds,  $f(s(x)) = blue$  holds. The protocol is given in Algorithm 1. Note that this algorithm does not care an initiator and a responder.

The basic strategy of the protocol is the following.

1. Create a spanning tree rooted at the base station. Concretely, agent  $x$  assigns its depth in a tree rooted at the base station into variable  $depth_x$ . Variable  $depth_x$  is initialized to  $\perp$ . Variable  $depth_x$  obtains the depth of  $x$  in the spanning tree as follows: If the base station and an agent  $p$  with  $depth_p = \perp$  interact,  $depth_p$  becomes 1. If an agent  $q$  with  $depth_q \neq \perp$  and an agent  $p$  with  $depth_p = \perp$  interact,  $depth_p$  becomes  $depth_q + 1$ . By these behaviors, for any agent  $x$ , eventually variable  $depth_x$  has a depth of  $x$  in a tree rooted at the base station.
2. Using the spanning tree, carry the initial color *ini* toward the base station and make the base station assign *r* and *b* to agents one by one. Concretely, if agents  $x$  and  $y$  interact

and both  $depth_y < depth_x$  and  $color_x = ini$  hold,  $x$  and  $y$  exchange their colors (i.e.,  $ini$  is carried from  $x$  to  $y$ ). Hence, since  $ini$  is always carried to a smaller  $depth$ , eventually an agent  $z$  with  $depth_z = 1$  obtains  $ini$ . After that, the base station and the agent  $z$  interact and the base station assigns  $r$  or  $b$  to  $z$ . Note that, if the base station assigns  $r$  (resp.,  $b$ ), the base station assigns  $b$  (resp.,  $r$ ) next.

Then, for any agent  $v$ , eventually  $color_v \neq ini$  holds. Hence, there exist  $\lceil n/2 \rceil$  red (resp., blue) agents, and  $\lfloor n/2 \rfloor$  blue (resp., red) agents if variable  $RB$  in the base station has  $r$  (resp.,  $b$ ) as an initial value. Therefore, the protocol solves the uniform bipartition problem.

► **Theorem 2.** *Algorithm 1 solves the uniform bipartition problem. That is, there exists a protocol with  $3P + 1$  states and designated initial states that solves the uniform bipartition problem under weak fairness assuming arbitrary communication graphs with a non-initialized base station.*

### 3.2.2 A protocol with constant states over a restricted class of graphs

In this subsection, we show that the space complexity of Algorithm 1 can be reduced to constant for communication graphs such that every cycle either includes the base station or its length is not a multiple of  $l$ , where  $l$  is a positive integer at least three.

We modify Algorithm 1 as follows. Each agent maintains the distance from the base station by computing modulo  $l$  plus 1. That is, we change lines 7 and 8 in Algorithm 1 to  $depth_x \leftarrow depth_y \bmod l + 1$  and  $depth_y \leftarrow depth_x \bmod l + 1$ , respectively. Now  $depth_x \in \{\perp, 1, 2, 3, \dots, l\}$  holds for any agent  $x$ . Then we redefine the relation  $depth_x < depth_y$  in lines 9 and 12 as follows:  $depth_x < depth_y$  holds if and only if either  $depth_x = 1 \wedge depth_y = 2$ ,  $depth_x = 2 \wedge depth_y = 3$ ,  $depth_x = 3 \wedge depth_y = 4$ ,  $\dots$ ,  $depth_x = l - 1 \wedge depth_y = l$ , or  $depth_x = l \wedge depth_y = 1$  holds.

We can easily observe that these modifications do not change the essence of Algorithm 1. For two agents  $x$  and  $y$ , we say  $x < y$  if  $depth_x < depth_y$  holds. Each agent  $x$  eventually assigns a depth of  $x$  modulo  $l$  plus 1 to  $depth_x$ , and at that time there exists a path  $x_0, x_1, \dots, x_h$  such that  $x_0$  is a neighbor of the base station,  $x = x_h$  holds, and  $x_i < x_{i+1}$  holds for any  $0 \leq i < h$ . In addition, there exists no cycle  $x_0, x_1, \dots, x_h = x_0$  such that  $x_i < x_{i+1}$  holds for any  $0 \leq i < h$ . This is because, from the definition of relation ' $<$ ', the length of such a cycle should be a multiple of  $l$ , but we assume that underlying communication graphs do not include a cycle of agents in  $V_p$  whose length is a multiple of  $l$ . Hence, similarly to Algorithm 1, we can carry the initial color  $ini$  toward the base station and make the base station assign  $r$  and  $b$  to agents one by one.

► **Corollary 3.** *There exists a protocol with  $3l + 1$  states and designated initial states that solves the uniform bipartition problem under weak fairness assuming arbitrary communication graphs with a non-initialized base station if, for any cycle of the communication graphs, it either includes the base station or its length is not a multiple of  $l$ , where  $l$  is a positive integer at least three.*

## 4 Upper and Lower Bounds with No Base Station

In this section, we show upper and lower bounds of the number of states to solve the uniform bipartition problem with no base station and designated initial states over arbitrary communication graphs. Concretely, under global fairness, we prove that the minimum number of states for asymmetric protocols is four, and the minimum number of states for symmetric protocols is five. Under weak fairness, we prove that the uniform bipartition problem cannot be solved without a base station using proof techniques similar to those Fischer and Jiang [21] used to show the impossibility of leader election.

■ **Algorithm 2** Transition rules of the uniform bipartition protocol with four states.

- 
1.  $(r^\omega, r^\omega) \rightarrow (r, b)$
  2.  $(r^\omega, b^\omega) \rightarrow (b, b)$
  3.  $(r^\omega, r) \rightarrow (r, r^\omega)$
  4.  $(b^\omega, b) \rightarrow (b, b^\omega)$
  5.  $(r^\omega, b) \rightarrow (r, b^\omega)$
  6.  $(b^\omega, r) \rightarrow (b, r^\omega)$
- 

#### 4.1 Upper Bound for Protocols under Global Fairness

In this subsection, over arbitrary graphs with designated initial states and no base station under global fairness, we give an asymmetric protocol with four states and a symmetric protocol with five states.

First, we show the asymmetric protocol with four states. We define a state set of agents as  $Q = \{r^\omega, b^\omega, r, b\}$ , and function  $f$  as follows:  $f(r^\omega) = f(r) = \text{red}$  and  $f(b^\omega) = f(b) = \text{blue}$ . We say an agent has a token if its state is  $r^\omega$  or  $b^\omega$ . Initially, every agent has state  $r^\omega$ , that is, every agent is *red* and has a token. The transition rules are given in Algorithm 2 (for each transition rule  $(p, q) \rightarrow (p', q')$  except for transition rule 1, transition rule  $(q, p) \rightarrow (q', p')$  exists, but we omit the description).

The basic strategy of the protocol is as follows. When two agents with tokens interact and one of them is *red*, a *red* agent transitions to *blue* and the two tokens are deleted (transition rules 1 and 2). Since  $n$  tokens exist initially and the number of tokens decreases by two in an interaction,  $\lfloor n/2 \rfloor$  *blue* agents appear and  $\lceil n/2 \rceil$  *red* agents remain after all tokens (except one token for the case of odd  $n$ ) disappear. To make such interactions, the protocol moves a token when agents with and without a token interact (transition rules 3, 4, 5, and 6). Global fairness guarantees that, if two tokens exist, an interaction of transition rule 1 or 2 happens eventually. Therefore, the uniform bipartition is achieved by the protocol.

► **Theorem 4.** *Algorithm 2 solves the uniform bipartition problem. That is, there exists a protocol with four states and designated initial states that solves the uniform bipartition problem under global fairness over arbitrary communication graphs.*

Furthermore, we obtain a symmetric protocol under the assumption by using a similar idea of the transformer proposed in [17]. The transformer simulates an asymmetric protocol on a symmetric protocol. To do this, the transformer requires additional states. Moreover, the transformer works with complete communication graphs. We show that one additional state is sufficient to transform the asymmetric uniform bipartition protocol into the symmetric protocol even if we assume arbitrary graphs (see the full version [34]).

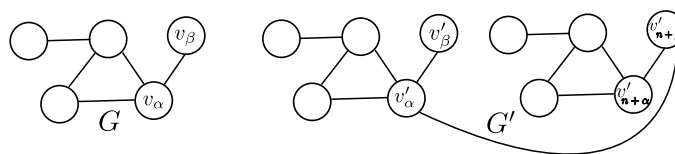
► **Theorem 5.** *There exists a symmetric protocol with five states and designated initial states that solves the uniform bipartition problem under global fairness with arbitrary communication graphs.*

#### 4.2 Lower Bound for Asymmetric Protocols under Global Fairness

In this section, we show that, over arbitrary graphs with designated initial states and no base station under global fairness, there exists no asymmetric protocol with three states.

To prove this, we first show that, when the number of agents  $n$  is odd and no more than  $P/2$ , each agent changes its own state to another state infinitely often in any globally-fair execution  $\Xi$  of a uniform bipartition protocol  $Alg$ , where  $P$  is a known upper bound of the number of agents. This lemma holds regardless of the number of states in a protocol.





■ **Figure 1** An example of communication graphs  $G$  and  $G'$  ( $n = 5$ ).

► **Lemma 6.** *Assume that there exists a uniform bipartition protocol  $Alg$  with designated initial states over arbitrary communication graphs assuming global fairness. Consider a graph  $G = (V, E)$  such that the number of agents  $n$  is odd and no more than  $P/2$ . In any globally-fair execution  $\Xi = C_0, C_1, \dots$  of  $Alg$  over  $G$ , each agent changes its state infinitely often.*

**Proof.** (Proof sketch) First, for the purpose of contradiction, we assume that there exists an agent  $v_\alpha$  that never changes its state after some stable configuration  $C_h$  in a globally-fair execution  $\Xi$  over graph  $G$ . Let  $s_\alpha$  be a state that  $v_\alpha$  has after  $C_h$ . Let  $v_\beta \in V$  be an agent adjacent to  $v_\alpha$  and  $S_\beta$  be a set of states that  $v_\beta$  has after  $C_h$ . Since the number of states is finite, there exists a stable configuration  $C_t$  that occurs infinitely often after  $C_h$ . Next, let  $G'_1 = (V'_1, E_1)$  and  $G'_2 = (V'_2, E_2)$  be graphs that are isomorphic to  $G$ . Moreover, let  $v'_\alpha \in V'_1$  (resp.,  $v'_{n+\beta} \in V'_2$ ) be an agent that corresponds to  $v_\alpha \in V$  (resp.,  $v_\beta \in V$ ). We construct  $G' = (V', E')$  by connecting  $G'_1$  and  $G'_2$  with an additional edge  $(v'_\alpha, v'_{n+\beta})$  (see Figure 1). Over  $G'$ , we consider an execution  $\Xi'$  such that, agents in  $G'_1$  and  $G'_2$  behave similarly to  $\Xi$  until  $C_t$  occurs in  $G'_1$  and  $G'_2$ , and then make interactions so that  $\Xi'$  satisfies global fairness. Since  $\Xi$  is globally-fair, we can show the following facts after  $G'_1$  and  $G'_2$  reach  $C_t$  in  $\Xi'$ .

- $v'_\alpha$  has state  $s_\alpha$  as long as  $v'_{n+\beta}$  has a state in  $S_\beta$ .
- $v'_{n+\beta}$  has a state in  $S_\beta$  as long as  $v'_\alpha$  has state  $s_\alpha$ .

From these facts, in  $\Xi'$ ,  $v'_\alpha$  continues to have state  $s_\alpha$  and  $v'_{n+\beta}$  continues to have a state in  $S_\beta$ . Hence, in  $\Xi'$ , each agent in  $V'_1$  cannot notice the existence of agents in  $V'_2$ , and vice versa. This implies that, in stable configurations,  $\#red(V) = \#red(V'_1) = \#red(V'_2)$  and  $\#blue(V) = \#blue(V'_1) = \#blue(V'_2)$  hold. Since the number of agents in  $G$  is odd,  $\#red(V) - \#blue(V) = 1$  or  $\#blue(V) - \#red(V) = 1$  holds in stable configurations of  $\Xi$ . Thus, in stable configurations of  $\Xi'$ ,  $|\#red(V') - \#blue(V')| = 2$  holds. Since  $\Xi'$  is globally-fair, this is a contradiction. ◀

Now we prove impossibility of an asymmetric protocol with three states. The outline of the proof is as follows. For the purpose of contradiction, we assume that there exists a protocol  $Alg$  that solves the problem with three states. From Lemma 6, in any globally-fair execution, some agents change their state infinitely often. Now, with three states, the number of *red* or *blue* states is at least one and thus, if we assume without loss of generality that the number of *blue* states is one, agents with the *blue* state change their color eventually after a stable configuration. This is a contradiction.

► **Theorem 7.** *There exists no uniform bipartition protocol with three states and designated initial states over arbitrary communication graphs assuming global fairness.*

### 4.3 Lower Bound for Symmetric Protocols under Global Fairness

In this section, we show that, with arbitrary communication graphs, designated initial states, and no base station assuming global fairness, there exists no symmetric protocol with four states. Recall that, with designated initial states and no base station, clearly any symmetric



### 33:12 Uniform Bipartition in the Population Protocol Model with Arbitrary Graphs

protocol never solves the problem if the number of agents  $n$  is two. Thus, we assume that  $3 \leq n \leq P$  holds, where  $P$  is a known upper bound of the number of agents. Note that the symmetric protocol proposed in subsection 4.1 solves the problem for  $3 \leq n \leq P$ .

► **Theorem 8.** *There exists no symmetric protocol for the uniform bipartition with four states and designated initial states over arbitrary graph assuming global fairness when  $P$  is twelve or more.*

For the purpose of contradiction, suppose that there exists such a protocol  $Alg$ . Let  $R$  (resp.,  $B$ ) be a state set such that, for any  $s \in R$  (resp.,  $s' \in B$ ),  $f(s) = red$  (resp.,  $f(s') = blue$ ) holds. First, we show that the following lemma holds from Lemma 6.

► **Lemma 9.**  $|R| = |B|$  holds (i.e.,  $|R| = 2$  and  $|B| = 2$  hold).

Let  $ini_r$  and  $r$  (resp.,  $ini_b$  and  $b$ ) be states belonging to  $R$  (resp.,  $B$ ). In addition, without loss of generality, assume that  $ini_r$  is the initial state of agents. Then, we can prove the following lemma.

► **Lemma 10.** *There exists some  $s_b \in B$  such that  $(ini_r, ini_r) \rightarrow (s_b, s_b)$  and  $(s_b, s_b) \rightarrow (ini_r, ini_r)$  hold.*

Without loss of generality, assume that  $(ini_r, ini_r) \rightarrow (ini_b, ini_b)$  and  $(ini_b, ini_b) \rightarrow (ini_r, ini_r)$  exist. For some population  $V$ , we denote the number of agents with  $ini_r$  (resp.,  $ini_b$ ) belonging to  $V$  as  $\#ini_r(V)$  (resp.,  $\#ini_b(V)$ ). Moreover, let  $\#ini(V)$  be the sum of  $\#ini_r(V)$  and  $\#ini_b(V)$ . When  $V$  is clear from the context, we simply denote them as  $\#ini_r$ ,  $\#ini_b$ , and  $\#ini$ , respectively. Then, we can prove the following lemmas and corollary.

► **Lemma 11.** *There does not exist a transition rule such that  $\#ini$  increases after the transition.*

► **Lemma 12.** *Consider a globally-fair execution  $\Xi$  of  $Alg$  with some complete communication graph  $G$ . After some configuration in  $\Xi$ ,  $\#ini \leq 1$  holds.*

► **Corollary 13.** *Consider a state set  $Ini = \{ini_r, ini_b\}$ . When  $s_1 \notin Ini$  or  $s_2 \notin Ini$  holds, if transition rule  $(s_1, s_2) \rightarrow (s'_1, s'_2)$  exists then  $f(s_1) = f(s'_1)$  and  $f(s_2) = f(s'_2)$  hold.*

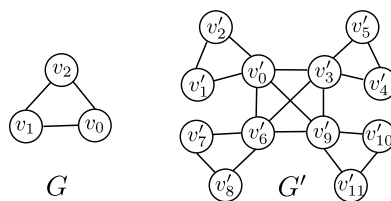
From now on, we prove Theorem 8. Consider a globally-fair execution  $\Xi = C_0, C_1, C_2, \dots$  of  $Alg$  with a ring communication graph  $G = (V, E)$  such that the number of agents is three, where  $V = \{v_0, v_1, v_2\}$ . In a stable configuration of  $\Xi$ , either  $\#blue(V) - \#red(V) = 1$  or  $\#red(V) - \#blue(V) = 1$  holds.

First, consider the case of  $\#blue(V) - \#red(V) = 1$ .

By Lemma 6,  $red$  agents keep exchanging  $r$  for  $ini_r$  in  $\Xi$ . Moreover, by Lemma 12, there exists a stable configuration in  $\Xi$  such that  $\#ini \leq 1$  holds. From these facts, there exists a stable configuration  $C_t$  of  $\Xi$  such that there exists exactly one agent that has  $ini_r$ . Without loss of generality, we assume that the agent is  $v_0$ .

Consider the communication graph  $G' = (V', E')$  that includes four copies of  $G$ . The details of  $G'$  are as follows:

- Let  $V' = \{v'_0, v'_1, v'_2, v'_3, \dots, v'_{11}\}$ . Moreover, we define a partition of  $V'$  as  $V'_1 = \{v'_0, v'_1, v'_2\}$ ,  $V'_2 = \{v'_3, v'_4, v'_5\}$ ,  $V'_3 = \{v'_6, v'_7, v'_8\}$ , and  $V'_4 = \{v'_9, v'_{10}, v'_{11}\}$ . Additionally, let  $V'_{red} = \{v'_0, v'_3, v'_6, v'_9\}$  be a set of agents that will have state  $ini_r$ .
- $E' = \{(v'_x, v'_y), (v'_{x+3}, v'_{y+3}), (v'_{x+6}, v'_{y+6}), (v'_{x+9}, v'_{y+9}) \in V' \times V' \mid (v_x, v_y) \in E\} \cup \{(v'_x, v'_y) \in V' \times V' \mid x, y \in \{0, 3, 6, 9\}\}$ .



■ **Figure 2** An image of graphs  $G$  and  $G'$ .

An image of  $G$  and  $G'$  is shown in Figure 2.

Consider the following execution  $\Xi' = C'_0, C'_1, C'_2, \dots$  of  $Alg$  with  $G' = (V', E')$ .

- For  $i \leq t$ , when  $v_x$  interacts with  $v_y$  at  $C_i \rightarrow C_{i+1}$ ,  $v'_x$  interacts with  $v'_y$  at  $C'_{4i} \rightarrow C'_{4i+1}$ ,  $v'_{x+3}$  interacts with  $v'_{y+3}$  at  $C'_{4i+1} \rightarrow C'_{4i+2}$ ,  $v'_{x+6}$  interacts with  $v'_{y+6}$  at  $C'_{4i+2} \rightarrow C'_{4i+3}$ , and  $v'_{x+9}$  interacts with  $v'_{y+9}$  at  $C'_{4i+3} \rightarrow C'_{4i+4}$ .
- After  $C'_{4t}$ , make interactions between agents in  $V'_{red}$  until agents in  $V'_{red}$  converge and  $\#ini(V'_{red}) \leq 1$  holds. We call the configuration  $C'_{t'}$ .
- After  $C'_{t'}$ , make interactions so that  $\Xi'$  satisfies global fairness.

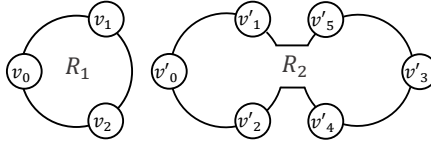
Until  $C'_{4t}$ , agents in  $V'_1, V'_2, V'_3$ , and  $V'_4$  behave similarly to agents in  $V$  from  $C_0$  to  $C_t$ . This implies that, in  $C'_{4t}$ , every agent in  $V'_{red}$  has state  $ini_r$ . From Lemma 12, since  $ini_r$  is the initial state of agents, it is possible to make interactions between agents in  $V'_{red}$  until agents in  $V'_{red}$  converge and  $\#ini(V'_{red}) \leq 1$  holds. Moreover, since  $v_0$  is the only agent that has  $ini_r$  in  $C_t$ , no agent in  $V'_i \setminus V'_{red} (1 \leq i \leq 4)$  has state  $ini_r$  or  $ini_b$  in  $C'_{4t}$ . Hence,  $\#ini \leq 1$  holds in  $C'_{t'}$ . By Corollary 13, if  $\#ini \geq 2$  does not hold, no agent can change its color. Thus, since  $\#ini \leq 1$  holds after  $C'_{t'}$  by Lemma 11, no agent can change its color after  $C'_{t'}$ . Since  $v_1$  and  $v_2$  are *blue* in  $C_t$ ,  $v'_1, v'_2, v'_4, v'_5, v'_7, v'_8, v'_{10}$ , and  $v'_{11}$  are *blue* in  $C'_{t'}$ . In addition,  $\#blue(V'_{red}) = \#red(V'_{red})$  holds. Hence,  $\#blue(V') - \#red(V') = 8$  holds. Since no agent can change its color after  $C'_{t'}$  and  $\Xi'$  is globally-fair, this is a contradiction.

Next, consider the case of  $\#red(V) - \#blue(V) = 1$ . In this case, we can prove in the same way as the case of  $\#blue(V) - \#red(V) = 1$ . However, in the case, we focus on  $ini_b$  instead of  $ini_r$ . That is, we assume that agents in  $V'_{red}$  (i.e.,  $v'_0, v'_3, v'_6$ , and  $v'_9$ ) have  $ini_b$  in  $C'_{4t}$ . From  $C'_{4t}$ , we make  $v'_0$  (resp.,  $v'_6$ ) interact with  $v'_3$  (resp.,  $v'_9$ ) once. Then, by Lemma 10, all of them transition to  $ini_r$ . After that, since all agents in  $V'_{red}$  have  $ini_r$ , we can construct an execution such that only agents in  $V'_{red}$  interact and eventually  $\#ini(V'_{red}) \leq 1$  holds. As a result, we can lead to contradiction in the same way as the case of  $\#blue(V) - \#red(V) = 1$ .

#### 4.4 Impossibility under Weak Fairness

In this subsection, assuming arbitrary communication graphs and designated initial states and no base station, we show that there is no protocol that solves the problem under weak fairness. Fischer and Jiang [21] proved the impossibility of leader election for a ring communication graph. We borrow their proof technique and apply it to the impossibility proof of a uniform bipartition problem.

The sketch of the proof is as follows: For the purpose of contradiction, let us assume that there exists such a protocol  $Alg$ . Consider an execution  $\Xi$  of  $Alg$  for a ring  $R_1$  with three agents  $v_0, v_1$ , and  $v_2$ . Without loss of generality, we assume that  $\#red = 1$  and  $\#blue = 2$  hold in a stable configuration of  $\Xi$ . After that, consider an execution  $\Xi'$  of  $Alg$  for a ring  $R_2$  with six agents  $v'_0, v'_1, v'_2, v'_3, v'_4$ , and  $v'_5$  (see Figure 3). We construct  $\Xi'$  such that each agent behaves similarly to  $\Xi$ . Concretely,  $v'_i$  and  $v'_{i+3} (0 \leq i \leq 3)$  behave similarly to  $v_i$ . If



■ **Figure 3** Ring graphs  $R_1$  and  $R_2$ .

$v_0$  interacts with  $v_1$  (resp.,  $v_2$ ) in  $\Xi$ ,  $v'_0$  interacts with  $v'_1$  (resp.,  $v'_2$ ) and  $v'_3$  interacts with  $v'_4$  (resp.,  $v'_5$ ) in  $\Xi'$ . Similarly, If  $v_1$  (resp.,  $v_2$ ) interacts with  $v_0$  in  $\Xi$ ,  $v'_1$  (resp.,  $v'_2$ ) interacts with  $v'_0$  and  $v'_4$  (resp.,  $v'_5$ ) interacts with  $v'_3$  in  $\Xi'$ . If  $v_1$  interacts with  $v_2$  in  $\Xi$ ,  $v'_1$  interacts with  $v'_5$  and  $v'_4$  interacts with  $v'_2$  in  $\Xi'$ . Similarly, if  $v_2$  interacts with  $v_1$  in  $\Xi$ ,  $v'_5$  interacts with  $v'_1$  and  $v'_2$  interacts with  $v'_4$  in  $\Xi'$ . Observe that, if  $s(v_i) = s(v'_i) = s(v'_{i+3})$  holds before the interactions for  $0 \leq i \leq 2$ ,  $s(v_i) = s(v'_i) = s(v'_{i+3})$  holds even after the interactions. Thus, since  $s(v_i) = s(v'_i) = s(v'_{i+3})$  holds in the initial configuration,  $s(v_i) = s(v'_i) = s(v'_{i+3})$  continues to hold. Hence, in the stable configuration of  $\Xi'$ ,  $\#red = 2$  and  $\#blue = 4$  hold. This contradicts that  $Alg$  solves the problem. Therefore, we have the following theorem.

► **Theorem 14.** *There exists no protocol that solves the uniform bipartition problem with designated initial states and no base station under weak fairness assuming arbitrary communication graphs.*

## 5 Concluding Remarks

In this paper, we consider the uniform bipartition problem with designated initial states assuming arbitrary communication graphs. We investigated the problem solvability, and even provided tight bounds (with respect to the number of states per agent) in the case of global fairness.

Our work raises interesting open problems:

- Is there a relation between the uniform bipartition problem and other classical problems such as counting, leader election, and majority? We pointed out the reuse of some proof arguments, but the existence of a more systematic approach is intriguing.
- What is the time complexity of the uniform bipartition problem?

---

## References

- 1 Dan Alistarh, James Aspnes, David Eisenstat, Rati Gelashvili, and Ronald L Rivest. Time-space trade-offs in population protocols. In *Proc. of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2560–2579, 2017.
- 2 Dan Alistarh, James Aspnes, and Rati Gelashvili. Space-optimal majority in population protocols. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2221–2239. SIAM, 2018.
- 3 Dan Alistarh and Rati Gelashvili. Polylogarithmic-time leader election in population protocols. In *Proc. of the 42nd International Colloquium on Automata, Languages, and Programming*, pages 479–491, 2015.
- 4 Dana Angluin, James Aspnes, Melody Chan, Michael J Fischer, Hong Jiang, and René Peralta. Stably computable properties of network graphs. In *Proc. of International Conference on Distributed Computing in Sensor Systems*, pages 63–74, 2005.
- 5 Dana Angluin, James Aspnes, Zoë Diamadi, Michael J Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed computing*, 18(4):235–253, 2006.

- 6 Dana Angluin, James Aspnes, and David Eisenstat. A simple population protocol for fast robust approximate majority. *Distributed Computing*, 21(2):87–102, 2008.
- 7 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.
- 8 James Aspnes, Joffroy Beauquier, Janna Burman, and Devan Sohler. Time and space optimal counting in population protocols. In *Proc. of International Conference on Principles of Distributed Systems*, pages 13:1–13:17, 2016.
- 9 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.
- 10 Joffroy Beauquier, Janna Burman, Simon Claviere, and Devan Sohler. Space-optimal counting in population protocols. In *Proc. of International Symposium on Distributed Computing*, pages 631–646, 2015.
- 11 Joffroy Beauquier, Julien Clement, Stephane Messika, Laurent Rosaz, and Brigitte Rozoy. Self-stabilizing counting in mobile sensor networks with a base station. In *Proc. of International Symposium on Distributed Computing*, pages 63–76, 2007.
- 12 Stav Ben-Nun, Tsvi Kopelowitz, Matan Kraus, and Ely Porat. An  $o(\log^{3/2} n)$  parallel time population protocol for majority with  $o(\log n)$  states. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, pages 191–199, 2020.
- 13 Petra Berenbrink, Robert Elsässer, Tom Friedetzky, Dominik Kaaser, Peter Kling, and Tomasz Radzik. A population protocol for exact majority with  $o(\log^{5/3} n)$  stabilization time and  $\theta(\log n)$  states. In *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*, volume 121 of *LIPICs*, pages 10:1–10:18, 2018.
- 14 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.
- 15 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.
- 16 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.
- 17 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.
- 18 Davide Canepa and Maria Gradinariu Potop-Butucaru. Self-stabilizing tiny interaction protocols. In *Proceedings of the Third International Workshop on Reliability, Availability, and Security*, page 10. ACM, 2010.
- 19 Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, and Eric Ruppert. When birds die: Making population protocols fault-tolerant. *Distributed Computing in Sensor Systems*, pages 51–66, 2006.
- 20 David Doty and David Soloveichik. Stable leader election in population protocols requires linear time. *Distributed Computing*, 31(4):257–271, 2018.
- 21 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.
- 22 Leszek Gąsieniec, David Hamilton, Russell Martin, Paul G Spirakis, and Grzegorz Stachowiak. Deterministic population protocols for exact majority and plurality. In *Proc. of International Conference on Principles of Distributed Systems*, pages 14:1–14:14, 2016.
- 23 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.

## 33:16 Uniform Bipartition in the Population Protocol Model with Arbitrary Graphs

- 24 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.
- 25 Adrian Kosowski and Przemyslaw Uznanski. Brief announcement: Population protocols are fast. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, pages 475–477, 2018.
- 26 Anissa Lamani and Masafumi Yamashita. Realization of periodic functions by self-stabilizing population protocols with synchronous handshakes. In *Proc. of International Conference on Theory and Practice of Natural Computing*, pages 21–33, 2016.
- 27 George B Mertzios, Sotiris E Nikolettseas, Christoforos L Raptopoulos, and Paul G Spirakis. Determining majority in networks with local interactions and very small local memory. In *International Colloquium on Automata, Languages, and Programming*, pages 871–882. Springer, 2014.
- 28 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.
- 29 Yuichi Sudo, Fukuhito Ooshita, Taisuke Izumi, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. Time-optimal leader election in population protocols. *IEEE Transactions on Parallel and Distributed Systems*, 2020.
- 30 Yuichi Sudo, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. Loosely-stabilizing leader election on arbitrary graphs in population protocols. In *International Conference on Principles of Distributed Systems*, pages 339–354. Springer, 2014.
- 31 Tomoki Umino, Naoki Kitamura, and Taisuke Izumi. Differentiation in population protocols. *6th workshop on biological distributed algorithms(BDA)*, 2018.
- 32 Hiroto Yasumi, Naoki Kitamura, Fukuhito Ooshita, Taisuke Izumi, and Michiko Inoue. A population protocol for uniform k-partition under global fairness. *International Journal of Networking and Computing*, 9(1):97–110, 2019.
- 33 Hiroto Yasumi, Fukuhito Ooshita, and Michiko Inoue. Uniform partition in population protocol model under weak fairness. *the 23rd International Conference on Principles of Distributed Systems*, 2019.
- 34 Hiroto Yasumi, Fukuhito Ooshita, Michiko Inoue, and Sébastien Tixeuil. Uniform bipartition in the population protocol model with arbitrary communication graphs, 2020. [arXiv:2011.08366](https://arxiv.org/abs/2011.08366).
- 35 Hiroto Yasumi, Fukuhito Ooshita, Ken’ichi Yamaguchi, and Michiko Inoue. Constant-space population protocols for uniform bipartition. *the 21st International Conference on Principles of Distributed Systems*, 2017.
- 36 Hiroto Yasumi, Fukuhito Ooshita, Ken’ichi Yamaguchi, and Michiko Inoue. Space-optimal population protocols for uniform bipartition under global fairness. *IEICE TRANSACTIONS on Information and Systems*, 102(3):454–463, 2019.