22nd International Conference on Principles of Distributed Systems

OPODIS 2018, December 17–19, 2018, Hong Kong, China

Edited by

Jiannong Cao
Faith Ellen
Luis Rodrigues
Bernardo Ferreira

Editors

Jiannong Cao
Department of Computing
The Hong Kong Polytechnic University
csjcao@comp.polyu.edu.hk

Faith Ellen
Department of Computer Science
University of Toronto
faith@cs.toronto.edu

Luis Rodrigues
Instituto Superior Técnico
Universidade de Lisboa
ler@tecnico.ulisboa.pt

Bernardo Ferreira
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
bf@fct.unl.pt

ACM Classification 2012
Computer systems organization → Dependable and fault-tolerant systems and networks, Computing methodologies → Distributed algorithms, Networks → Mobile networks, Wireless access networks, Ad hoc networks, Software and its engineering → Distributed systems organizing principles, Theory of computation → Distributed computing models, Theory of computation → Data structures design and analysis, Theory of computation → Distributed algorithms

ISBN 978-3-95977-098-9

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at http://www.dagstuhl.de/dagpub/978-3-95977-098-9.

Publication date
January, 2019

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

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

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

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

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPIcs.OPODIS.2018.0

ISBN 978-3-95977-098-9 **ISSN 1868-8969** **http://www.dagstuhl.de/lipics**

LIPIcs – Leibniz International Proceedings in Informatics

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

Editorial Board

- Luca Aceto (*Chair*, Gran Sasso Science Institute and Reykjavik University)
- Susanne Albers (TU München)
- Christel Baier (TU Dresden)
- Javier Esparza (TU München)
- Michael Mitzenmacher (Harvard University)
- Madhavan Mukund (Chennai Mathematical Institute)
- Anca Muscholl (University Bordeaux)
- Catuscia Palamidessi (INRIA)
- Raimund Seidel (Saarland University and Schloss Dagstuhl – Leibniz-Zentrum für Informatik)
- Thomas Schwentick (TU Dortmund)
- Reinhard Wilhelm (Saarland University)

ISSN 1868-8969

http://www.dagstuhl.de/lipics

Contents

Front Matter

Preface
Jiannong Cao, Faith Ellen, and Luis Rodrigues 0:ix–0:x

Program Committee
.... 0:xi–0:xii

Steering Committee
.... 0:xiii

Organization Committee
.... 0:xv

List of Authors
.... 0:xvii–0:xx

Keynotes

Complexity of Multi-Valued Register Simulations: A Retrospective
Jennifer L. Welch 1:1–1:1

Distributed Systems and Databases of the Globe Unite! The Cloud, the Edge and Blockchains
Amr El Abbadi 2:1–2:1

How to Make Decisions (Optimally)
Siddhartha Sen 3:1–3:1

Regular Papers

Session 1:

Sparse Matrix Multiplication and Triangle Listing in the Congested Clique Model
Keren Censor-Hillel, Dean Leitersdorf, and Elia Turner 4:1–4:17

Large-Scale Distributed Algorithms for Facility Location with Outliers
Tanmay Inamdar, Shreyas Pai, and Sriram V. Pemmaraju 5:1–5:16

Equilibria of Games in Networks for Local Tasks
Simon Collet, Pierre Fraigniaud, and Paolo Penna 6:1–6:16

The Sparsest Additive Spanner via Multiple Weighted BFS Trees
Keren Censor-Hillel, Ami Paz, and Noam Ravid 7:1–7:16

Session 2:

The Amortized Analysis of a Non-blocking Chromatic Tree
Jeremy Ko 8:1–8:17

22nd International Conference on Principles of Distributed Systems (OPODIS 2018).
Editors: Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
OPODIS

Lock-Free Search Data Structures: Throughput Modeling with Poisson Processes
Aras Atalar, Paul Renaud-Goud, and Philippas Tsigas 9:1–9:16

Concurrent Robin Hood Hashing
Robert Kelly, Barak A. Pearlmutter, and Phil Maguire 10:1–10:16

Session 3:

Parallel Combining: Benefits of Explicit Synchronization
Vitaly Aksenov, Petr Kuznetsov, and Anatoly Shalyto 11:1–11:16

Specification and Implementation of Replicated List: The Jupiter Protocol Revisited
Hengfeng Wei, Yu Huang, and Jian Lu 12:1–12:16

Local Fast Segment Rerouting on Hypercubes
Klaus-Tycho Foerster, Mahmoud Parham, Stefan Schmid, and Tao Wen 13:1–13:17

Session 4:

Effects of Topology Knowledge and Relay Depth on Asynchronous Appoximate Consensus
Dimitris Sakavalas, Lewis Tseng, and Nitin H. Vaidya 14:1–14:16

Hybrid Fault-Tolerant Consensus in Asynchronous and Wireless Embedded Systems
Wenbo Xu, Signe Rüsch, Bijun Li, and Rüdiger Kapitza 15:1–15:16

Correctness of Tendermint-Core Blockchains
Yackolley Amoussou-Guenou, Antonella Del Pozzo, Maria Potop-Butucaru, and Sara Tucci-Piergiovanni 16:1–16:16

Federated Byzantine Quorum Systems
Álvaro García-Pérez and Alexey Gotsman 17:1–17:16

Session 5:

Characterizing Asynchronous Message-Passing Models Through Rounds
Adam Shimi, Aurélie Hurault, and Philippe Quéinnec 18:1–18:17

You Only Live Multiple Times: A Blackbox Solution for Reusing Crash-Stop Algorithms In Realistic Crash-Recovery Settings
David Kozhaya, Ognjen Marić, and Yvonne-Anne Pignolet 19:1–19:17

Causal Broadcast: How to Forget?
Brice Nédelec, Pascal Molli, and Achour Mostéfaoui 20:1–20:16

Session 6:

Output-Oblivious Stochastic Chemical Reaction Networks
Ben Chugg, Hooman Hashemi, and Anne Condon 21:1–21:16

The Synergy of Finite State Machines
Yehuda Afek, Yuval Emek, and Noa Kolikant 22:1–22:16

Task Computability in Unreliable Anonymous Networks
Petr Kuznetsov and Nayuta Yanagisawa 23:1–23:13

Session 7:

Optimal Rendezvous $\mathcal{L}$-Algorithms for Asynchronous Mobile Robots with External-Lights
Takashi Okumura, Koichi Wada, and Xavier Défago 24:1–24:16

Linear Rendezvous with Asymmetric Clocks
Jurek Czyzowicz, Ryan Killick, and Evangelos Kranakis 25:1–25:16

Approximate Neighbor Counting in Radio Networks
Calvin Newport and Chaodong Zheng 26:1–26:16

On Simple Back-Off in Unreliable Radio Networks
Seth Gilbert, Nancy Lynch, Calvin Newport, and Dominik Pajak 27:1–27:17

Session 8:

Concurrent Specifications Beyond Linearizability
Éric Goubault, Jérémy Ledent, and Samuel Mimram 28:1–28:16

Parameterized Synthesis of Self-Stabilizing Protocols in Symmetric Rings
Nahal Mirzaie, Fathiyeh Faghih, Swen Jacobs, and Borzoo Bonakdarpour 29:1–29:17

Loosely-Stabilizing Leader Election with Polylogarithmic Convergence Time
Yuichi Sudo, Fukuhito Ooshita, Hirotsugu Kakugawa, Toshimitsu Masuzawa, Ajoy K. Datta, and Lawrence L. Larmore 30:1–30:16

Self-Stabilizing Token Distribution with Constant-Space for Trees
Yuichi Sudo, Ajoy K. Datta, Lawrence L. Larmore, and Toshimitsu Masuzawa 31:1–31:16

Preface

The papers in this volume were presented at the 22nd International Conference on Principles of Distributed Systems (OPODIS 2018), held on December 17-19, 2018, in Hong Kong, China. The conference was organized by the Hong Kong Polytechnic University (PolyU).

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 whole range between the theoretical aspects and practical implementations of distributed systems, as well as experimentation 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
- Dependable distributed algorithms and systems
- Design and analysis of distributed data structures
- Design and analysis of distributed graph algorithms
- Distributed operating systems and middleware
- Embedded systems
- Formal methods
- Game-theory and economical aspects of distributed computing
- Impossibility results for distributed computing
- Mesh and ad-hoc networks
- Mobile agents, robots, and rendezvous
- Randomization in distributed computing
- Security and privacy
- Self-stabilization
- Shared memory algorithms
- Specification and verification of distributed systems
- Synchronization
- Transactional memory

We received 66 submissions, each of which was reviewed 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 66 submissions, 28 papers were selected to be included in these proceedings.

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

The Best Paper Award was given to Seth Gilbert, Nancy Lynch, Calvin Newport, and Dominik Pajak, for their paper entitled “On Simple Back-Off in Unreliable Radio Networks”. The Best Student Paper Award was given to Dean Leitersdorf and and Elia Turner for their paper entitled “Sparse Matrix Multiplication and Triangle Listing in the Congested Clique Model”, coauthored with Keren Censor-Hillel. Some of the papers were selected by the Program Committee for a special issue of Theoretical Computer Science.

This year OPODIS had three distinguished invited keynote speakers: Amr El Abbadi (University of California Santa Barbara, USA), Siddhartha Sen (Microsoft Research New York City, USA), and Jennifer Welch (Texas A&M University, USA)

We would like to thank all the authors for submitting their work to OPODIS. We are also grateful to the members of the Program Committee for their hard work reviewing papers and their active participation in the online discussions and the program committee meeting held via teleconference. We also thank the external reviewers for their help with the reviewing process.

Organizing this event would not have been possible without the time and the effort of Carmen Au, the local arrangements chair, Jia Wang, who was responsible for the website, and Bernardo Ferreira, who managed the proceedings.

Finally, we would like to thank the Steering Committee, especially Sebastien Tixeuil, for their valuable advice and to Hong Kong Polytechnic University for their support.

December 2018

Jiannong Cao (Hong Kong Polytechnic University)
Faith Ellen (University of Toronto)
Luís Rodrigues (INESC-ID, Instituto Superior Técnico, Universidade de Lisboa)

Program Committee

General Chair

Jiannong Cao, The Hong Kong Polytechnic University, Hong Kong

Program Chairs

Faith Ellen, University of Toronto, Canada

Luis Rodrigues, INESC-ID, IST, Universidade de Lisboa, Portugal

Program Committee

Dan Alistarh, Institute of Science and Technology, Austria

Lorenzo Alvisi, Cornell University, USA

Alysson Bessani, Faculdade de Ciências, Universidade de Lisboa, Portugal

François Bonnet, Tokyo Institute of Technology, Japan

Manuel Bravo, Université Catholique de Louvain, Belgium

Janna Burman, Université Paris-Saclay, France

Christian Cachin, IBM Research, Zurich, Switzerland

Marco Canini, KAUST, Saudi Arabia

Lydia Chen, TU Delft, Netherlands

Michael Dinitz, Johns Hopkins, USA

Pedro Fonseca, University of Washington

Roy Friedman, Technion, Israel

George Giakkoupis, Inria, France

Seth Gilbert, National University of Singapore, Singapore

Magnus Halldorsson, Reykjavik University, Iceland

Michiko Inoue, Nara Institute of Science and Technology, Japan

Valerie Issarny, Inria, France

Taisuke Izumi, Nagoya Institute of Technology, Japan

Vasiliki Kalavri, ETH, Zurich

João Leitão, NOVA LINCS & FCT, Universidade Nova de Lisboa, Portugal

Avery Miller, University of Manitoba, Canada

Mark Moir, Oracle Labs

Alberto Montresor, University of Trento, Italy

Adam Morrison, Tel-Aviv University, Israel

Lata Narayanan, Concordia University, Canada

Vivien Quema, Grenoble INP, France

Marco Serafini, Qatar Computing Research Institute, Qatar

Michael Spear, Lehigh University, USA

Corentin Travers, Université de Bordeaux, France

Jennifer Welch, Texas A&M University, USA

Josef Widder, Technische Universität Wien, Austria

Yongluan Zhou, University of Copenhagen, Denmark

Steering Committee

James Aspres, Yale University, USA

Panagiota Fatourou, FORTH ICS & University of Crete, Greece

Pascal Felber, University of Neuchâtel, Switzerland

Alessia Milani, University of Bordeaux, France

Fernando Pedone, University of Lugano, Switzerland

Etienne Rivière, Université Catholique de Louvain, Belgium

Yukiko Yamauchi, University of Kyushu, Japan

Sebastien Tixeuil, IUF & Université Pierre et Marie Curie - Paris 6, France (Chair)

22nd International Conference on Principles of Distributed Systems (OPODIS 2018).
Editors: Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
OPODIS

Organization Committee

Carmen Au (Local Organization Chair), The Hong Kong Polytechnic University, Hong Kong

Jia Wang (Web Chair), The Hong Kong Polytechnic University, Hong Kong

Bernardo Ferreira (Proceedings Chair), NOVA LINCS & FCT, Universidade Nova de Lisboa, Portugal

List of Authors

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

Vitaly Aksenov
ITMO University, Saint-Petersburg, Russia
Inria, Paris, France
aksenov@corp.ifmo.ru

Yackolley Amoussou-Guenou
Institut LIST, CEA, Université Paris-Saclay, Palaiseau, France
Sorbonne Université, CNRS, Laboratoire d'Informatique de Paris 6, Paris, France

Aras Atalar
Chalmers University of Technology, S-41296 Göteborg, Sweden
aaras@chalmers.se

Borzoo Bonakdarpour
Iowa State University, 207 Atanasoff Hall, Ames, USA
borzoo@iastate.edu

Keren Censor-Hillel
Department of Computer Science, Technion, Israel
ckeren@cs.technion.ac.il

Ben Chugg
The University of British Columbia, Canada
ben.ih.chugg@gmail.com

Simon Collet
CNRS and University Paris Diderot, France

Anne Condon
The University of British Columbia, Canada
condon@cs.ubc.ca

Jurek Czyzowicz
Département d'informatique, Université du Québec en Outaouais, Canada

Ajoy K. Datta
Department of Computer Science, University of Nevada, Las Vegas,USA
ajoy.datta@unlv.edu

Xavier Défago
School of Computing, Tokyo Institute of Technology, Tokyo, Japan
defago@c.titech.ac.jp

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

Fathiyeh Faghih
University of Tehran, North Kargar St., Tehran, Iran
f.faghih@ut.ac.ir

Klaus-Tycho Foerster
University of Vienna, Vienna, Austria
klaus-tycho.foerster@univie.ac.at

Pierre Fraigniaud
CNRS and University Paris Diderot, France

Álvaro García-Pérez
IMDEA Software Institute, Madrid, Spain

Seth Gilbert
National University of Singapore, Singapore
seth.gilbert@comp.nus.edu.sg

Éric Goubault
École Polytechnique, Palaiseau, France
eric.goubault@lix.polytechnique.fr

Alexey Gotsman
IMDEA Software Institute, Madrid, Spain

Hooman Hashemi
The University of British Columbia, Canada
hhoomn390@gmail.com

Yu Huang
State Key Laboratory for Novel Software Technology, Nanjing University, China
yuhuang@nju.edu.cn

Aurélie Hurault
IRIT – Université de Toulouse, Toulouse, France
aurelie.hurault@irit.fr

22nd International Conference on Principles of Distributed Systems (OPODIS 2018).
Editors: Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
OPODIS

Tanmay Inamdar
Department of Computer Science, The University of Iowa, Iowa, USA
tanmay-inamdar@uiowa.edu

Swen Jacobs
CISPA Helmholtz Center i.G., Saarbrücken,Germany
jacobs@cispa.saarland

Hirotsugu Kakugawa
Graduate School of Information Science and Technology, Osaka University, Japan
kakugawa@ist.osaka-u.ac.jp

Rüdiger Kapitza
Technische Universität Braunschweig, Braunschweig, Germany
kapitza@ibr.cs.tu-bs.de

Robert Kelly
Maynooth University Department of Computer Science, Maynooth, Ireland
rob.kelly@cs.nuim.ie

Ryan Killick
School of Computer Science, Carleton University, Ottawa, Canada

Jeremy Ko
Department of Computer Science, University of Toronto, Canada
jerko@cs.toronto.edu

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

David Kozhaya
ABB Corporate Research, Switzerland
david.kozhaya@ch.abb.com

Evangelos Kranakis
School of Computer Science, Carleton University, Ottawa, Canada

Petr Kuznetsov
LTCI, Télécom ParisTech, Université Paris-Saclay, Paris, France
petr.kuznetsov@telecom-paristech.fr

Lawrence L. Larmore
Department of Computer Science, University of Nevada, Las Vegas,USA
lawrence.larmore@unlv.edu

Jérémy Ledent
École Polytechnique, Palaiseau, France
jeremy.ledent@lix.polytechnique.fr

Dean Leitersdorf
Department of Computer Science, Technion, Israel
dean.leitersdorf@gmail.com

Bijun Li
Technische Universität Braunschweig, Braunschweig, Germany
bli@ibr.cs.tu-bs.de

Jian Lu
State Key Laboratory for Novel Software Technology, Nanjing University, China
lj@nju.edu.cn

Nancy Lynch
MIT, Cambridge, MA, USA
lynch@csail.mit.edu

Phil Maguire
Maynooth University Department of Computer Science, Maynooth, Ireland
pmaguire@cs.nuim.ie

Ognjen Marić
Digital Asset, Switzerland
ogi.yolmt@mynosefroze.com

Toshimitsu Masuzawa
Graduate School of Information Science and Technology, Osaka University, Japan
masuzawa@ist.osaka-u.ac.jp

Samuel Mimram
École Polytechnique, Palaiseau, France
samuel.mimram@lix.polytechnique.fr

Nahal Mirzaie
University of Tehran, North Kargar St., Tehran, Iran
mirzaienahal@ut.ac.ir

Pascal Molli
LS2N, University of Nantes, France
pascal.molli@ls2n.fr

Achour Mostéfaoui
LS2N, University of Nantes, France
achour.mostefaoui@ls2n.fr

Brice Nédelec
LS2N, University of Nantes, France
brice.nedelec@ls2n.fr

Calvin Newport
Georgetown University, Washington, D.C., United States
cnewport@cs.georgetown.edu

Takashi Okumura
Graduate School of Science and Engineering, Hosei University, Tokyo, Japan
takashi.okumura.4e@stu.hosei.ac.jp

Fukuhito Ooshita
Graduate School of Science and Technology, Nara Institute of Science and Technology, Japan
f-oosita@is.naist.jp

Shreyas Pai
Department of Computer Science, The University of Iowa, Iowa, USA
shreyas-pai@uiowa.edu

Dominik Pajak
MIT, Cambridge, MA, USA
pajak@csail.mit.edu

Mahmoud Parham
University of Vienna, Vienna, Austria
mahmoud.parham@univie.ac.at

Ami Paz
IRIF, CNRS and Paris Diderot University, Paris, France
amipaz@irif.fr

Barak A. Pearlmutter
Maynooth University Department of Computer Science and Hamilton Institute, Maynooth, Ireland

Sriram V. Pemmaraju
Department of Computer Science, The University of Iowa, Iowa, USA
sriram-pemmaraju@uiowa.edu

Paolo Penna
ETH Zurich, Switzerland

Yvonne-Anne Pignolet
ABB Corporate Research, Switzerland
yvonne-anne.pignolet@ch.abb.com

Maria Potop-Butucaru
Sorbonne Université, CNRS, Laboratoire d'Informatique de Paris 6, Paris, France

Antonella Del Pozzo
Institut LIST, CEA, Université Paris-Saclay, Palaiseau, France

Philippe Quéinnec
IRIT – Université de Toulouse, Toulouse, France
philippe.queinnec@irit.fr

Noam Ravid
Department of Computer Science, Technion, Haifa, Israel
noamrvd@cs.technion.ac.il

Paul Renaud-Goud
Informatics Research Institute of Toulouse, F-31062 Toulouse, France
Paul.Renaud.Goud@irit.fr

Signe Rüsch
Technische Universität Braunschweig, Braunschweig, Germany
ruesch@ibr.cs.tu-bs.de

Dimitris Sakavalas
Boston College, USA
dimitris.sakavalas@bc.edu

Stefan Schmid
University of Vienna, Vienna, Austria
stefan_schmid@univie.ac.at

Anatoly Shalyto
ITMO University, Saint-Petersburg, Russia
shalyto@mail.ifmo.ru

Adam Shimi
IRIT – Université de Toulouse, Toulouse, France
adam.shimi@irit.fr

Yuichi Sudo
Graduate School of Information Science and Technology, Osaka University, Japan
y-sudou@ist.osaka-u.ac.jp

Lewis Tseng
Boston College, USA
lewis.tseng@bc.edu

Philippas Tsigas
Chalmers University of Technology, S-41296
Göteborg, Sweden
philippas.tsigas@chalmers.se

Sara Tucci-Piergiovanni
Institut LIST, CEA, Université Paris-Saclay,
Palaiseau, France

Elia Turner
Department of Computer Science, Technion,
Israel
eliaturner11@gmail.com

Nitin H. Vaidya
Georgetown University, USA
nitin.vaidya@georgetown.edu

Koichi Wada
Faculty of Science and Engineering, Hosei
University, Tokyo, Japan
wada@hosei.ac.jp

Hengfeng Wei
State Key Laboratory for Novel Software
Technology, Nanjing University, China
hfwei@nju.edu.cn

Tao Wen
University of Electronic Science and
Technology of China, Chengdu, China
winterwentao@gmail.com

Wenbo Xu
Technische Universität Braunschweig,
Braunschweig, Germany
wxu@ibr.cs.tu-bs.de

Nayuta Yanagisawa
DeNA Co., Ltd., Japan
nayuta.yanagisawa@dena.com

Chaodong Zheng
State Key Laboratory for Novel Software
Technology, Nanjing University, Nanjing,
China
chaodong@nju.edu.cn

Complexity of Multi-Valued Register Simulations: A Retrospective

Jennifer L. Welch
Department of Computer Science and Engineering, Texas A&M University, USA
welch@cse.tamu.edu

Abstract

I will provide a historical perspective on wait-free simulations of multi-bit shared registers using single-bit shared registers, starting with classical results from the last century and ending with an overview of the recent resurgence of interest in the topic. Particular emphasis will be placed on the space and step complexities of such simulations.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Distributed Systems

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.1

Category Keynote

© Jennifer L. Welch;
licensed under Creative Commons License CC-BY
22nd International Conference on Principles of Distributed Systems (OPODIS 2018).
Editors: Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira; Article No. 1; pp. 1:1–1:1

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

Distributed Systems and Databases of the Globe Unite! The Cloud, the Edge and Blockchains

Amr El Abbadi
Department of Computer Science, University of California, Santa Barbara, USA
amr@cs.ucsb.edu

Abstract

Significant paradigm shifts are occurring in Access patterns are widely dispersed and large scale analysis requires real-time responses. Many of the fundamental challenges have been studied and explored by both the distributed systems and the database communities for decades. However, the current changing and scalable setting often requires a rethinking of basic assumptions and premises. The rise of the cloud computing paradigm with its global reach has resulted in novel approaches to integrate traditional concepts in novel guises to solve fault-tolerance and scalability challenges. This is especially the case when users require real-time global access. Exploiting edge cloud resources becomes critical for improved performance, which requires a reevaluation of many paradigms, even for a traditional problem like caching. The need for transparency and accessibility has led to innovative ways for managing large scale replicated logs and ledgers, giving rise to blockchains and their many applications. In this talk we will be explore some of these new trends while emphasizing the novel challenges they raise from both distributed systems as well as database points of view. We will propose a unifying framework for traditional consensus and commitment protocols, and discuss novel protocols that exploit edge computing resources to enhance performance. We will highlight the advantages and discuss the limitations of blockchains. Our overall goal is to explore approaches that unite and exploit many of the significant efforts made in distributed systems and databases to address the novel and pressing needs of today's global computing infrastructure.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Consensus, Commitment, Cloud, Edge Computing, Blockchain

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.2

Category Keynote

© Amr El Abbadi;
licensed under Creative Commons License CC-BY
22nd International Conference on Principles of Distributed Systems (OPODIS 2018).
Editors: Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira; Article No. 2; pp. 2:1–2:1

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

How to Make Decisions (Optimally)

Siddhartha Sen
Microsoft Research New York City, USA
sidsen@microsoft.com

Abstract

Distributed systems are constantly faced with difficult decisions to make, such as in scheduling, caching, and traffic routing, to name a few. In most of these scenarios, the optimal decision is unknown and depends heavily on context. How can a system designer know if they have deployed the best decision-making policy, or if a different policy would perform better? As a community, we have developed a few methodologies for answering this question, some of them offline (e.g., simulation, trace-driven modeling) and some of them online (e.g., A/B testing). Neither approach is satisfactory: the offline methods suffer from bias and rely heavily on domain knowledge; the online methods are costly and difficult to deploy. What system designers ideally seek is the ability to ask "what if" questions about a policy without ever deploying it, which is called *counterfactual evaluation*. In this talk, I will show how reinforcement learning and causal inference can be synthesized to counterfactually evaluate a distributed system. We will apply this methodology to infrastructure systems in Azure, and face fundamental challenges and opportunities along the way. This talk will serve as an introduction to reinforcement learning and the counterfactual way of thinking, which I hope will interest and inspire the OPODIS community.

I will start by introducing reinforcement learning (RL) as the right framework for modeling decisions in a distributed system. In RL, an agent learns by interacting with its environment: i.e., making decisions and receiving feedback for them. This is a stark contrast to traditional (supervised) learning, where the correct answer, or "label", is known. Since an RL agent does not know the correct answer, it must constantly explore its world by randomizing some of its decisions. Now it turns out that this randomization, if used correctly, can give us a special superpower: the ability to *evaluate policies that have never been deployed*. As magical as this may sound, we can use statistics to show that this evaluation is indeed correct.

Unfortunately, applying this methodology to distributed systems is far from straightforward. Systems are complex, stateful amalgamations of components that navigate large decision spaces. We will need to wear both an RL hat and a systems hat to address these challenges. On the other hand, systems also present exciting opportunities. Many systems already use randomization in their decisions, e.g., to distribute data or work over replicas, or to manage resource contention. Sometimes, a conservative decision can implicitly yield feedback for other decisions: for example, when waiting for a timeout to expire, we automatically get feedback for what would have happened if we waited for any shorter amount of time. I will show how we can harvest this randomness and implicit feedback to achieve more effective counterfactual evaluation.

We will apply all of the above ideas to two production infrastructure systems in Azure: a machine health monitor that decides when to reboot unresponsive machines, and a geo-distributed edge proxy that chooses the TCP configuration of each proxy machine. In both cases, we are able to counterfactually evaluate arbitrary policies with estimates that match the ground truth. Production environments raise interesting constraints and challenges, some of which are preventing us from scaling up our methodology. I will describe a possible path forward, and invite others in the community to contemplate these problems as well.

2012 ACM Subject Classification Computing methodologies → Reinforcement learning, Software and its engineering → Software organization and properties

Keywords and phrases reinforcement learning, distributed systems, counterfactual evaluation

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.3

Category Keynote

© Siddhartha Sen;
licensed under Creative Commons License CC-BY
22nd International Conference on Principles of Distributed Systems (OPODIS 2018).
Editors: Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira; Article No. 3; pp. 3:1–3:1

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

Sparse Matrix Multiplication and Triangle Listing in the Congested Clique Model

Keren Censor-Hillel
Department of Computer Science, Technion, Israel
ckeren@cs.technion.ac.il

Dean Leitersdorf
Department of Computer Science, Technion, Israel
dean.leitersdorf@gmail.com

Elia Turner
Department of Computer Science, Technion, Israel
eliaturner11@gmail.com

Abstract

We show how to multiply two $n \times n$ matrices S and T over semirings in the CONGESTED CLIQUE model, where n nodes communicate in a fully connected synchronous network using $O(\log n)$-bit messages, within $O(nz(S)^{1/3}nz(T)^{1/3}/n + 1)$ rounds of communication, where $nz(S)$ and $nz(T)$ denote the number of non-zero elements in S and T, respectively. By leveraging the sparsity of the input matrices, our algorithm greatly reduces communication costs compared with general multiplication algorithms [Censor-Hillel et al., PODC 2015], and thus improves upon the state-of-the-art for matrices with $o(n^2)$ non-zero elements. Moreover, our algorithm exhibits the additional strength of surpassing previous solutions also in the case where only one of the two matrices is such. Particularly, this allows to efficiently raise a sparse matrix to a power greater than 2. As applications, we show how to speed up the computation on non-dense graphs of 4-cycle counting and all-pairs-shortest-paths.

Our algorithmic contribution is a new *deterministic* method of restructuring the input matrices in a sparsity-aware manner, which assigns each node with element-wise multiplication tasks that are not necessarily consecutive but guarantee a balanced element distribution, providing for communication-efficient multiplication.

Moreover, this new deterministic method for restructuring matrices may be used to restructure the adjacency matrix of input graphs, enabling faster deterministic solutions for graph related problems. As an example, we present a new sparsity aware, *deterministic* algorithm which solves the triangle listing problem in $O(m/n^{5/3} + 1)$ rounds, a complexity that was previously obtained by a *randomized* algorithm [Pandurangan et al., SPAA 2018], and that matches the known lower bound of $\tilde{\Omega}(n^{1/3})$ when $m = n^2$ of [Izumi and Le Gall, PODC 2017, Pandurangan et al., SPAA 2018]. Naturally, our triangle listing algorithm also implies triangle counting within the same complexity of $O(m/n^{5/3} + 1)$ rounds, which is (possibly more than) a *cubic* improvement over the previously known *deterministic* $O(m^2/n^3)$-round algorithm [Dolev et al., DISC 2012].

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis, Theory of computation → Distributed algorithms

Keywords and phrases congested clique, matrix multiplication, triangle listing

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.4

Related Version Some proofs are omitted from this paper and are presented in the full version, available online at [8], `https://arxiv.org/abs/1802.04789`.

© Keren Censor-Hillel, Dean Leitersdorf, and Elia Turner;
licensed under Creative Commons License CC-BY
22nd International Conference on Principles of Distributed Systems (OPODIS 2018).
Editors: Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira; Article No. 4; pp. 4:1–4:17

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

Funding This project has received funding from the European Union's Horizon 2020 Research And Innovation Programe under grant agreement no. 755839. Supported in part by ISF grant 1696/14.

Acknowledgements The authors thank Seri Khoury, Christoph Lenzen, and Merav Parter for many useful discussions and suggestions.

1 Introduction

Matrix multiplication is a fundamental algebraic task, with abundant applications to various computations. The value of the exponent ω of matrix multiplication, that is, the value ω for which $\Theta(n^\omega)$ is the complexity of matrix multiplication, is a central question in algebraic algorithms [25, 9, 26], and is currently known to be bounded by 2.3728639 [13].

The work of Censor-Hillel et al. [7] recently showed that known matrix multiplication algorithms for the *parallel* setting can be adapted to the distributed Congested Clique model, which consists of n nodes in a fully connected synchronous network, limited by a bandwidth of $O(\log n)$ bits per message. Subsequently, this significantly improved the state-of-the-art for a variety of tasks, including triangle and 4-cycle counting, girth computations, and (un)weighted/(un)directed all-pairs-shortest-paths (APSP). This was followed by the beautiful work of Le Gall [14], who showed how to efficiently multiply rectangular matrices, as well as multiple independent multiplication instances. These led to even faster algorithms for some of the tasks, such as weighted or directed APSP, as well as fast algorithms for new tasks, such as computing the size of the maximum matching.

In many cases, multiplication is required to be carried out for *sparse* matrices, and this need has been generating much effort in designing algorithms that are faster given sparse inputs, both in sequential (e.g., [27, 12, 17, 1, 15]) and parallel (e.g., [3, 5, 6, 2, 4, 20, 19, 24]) settings.

In this paper we focus our attention on the task of multiplying sparse matrices in the Congested Clique model, providing a novel *deterministic* algorithm with a round complexity which depends on the sparsity of the input matrices.

An immediate application of our algorithm is faster counting of 4-cycles. Moreover, a prime feature of our algorithm is that it speeds up matrix multiplication even if *only one* of the input matrices is sparse. The significance of this ability stems from the fact that the product of sparse matrices may be non-sparse, which in general may stand in the way of fast multiplication of more than two sparse matrices, such as raising a sparse matrix to a power that is larger than 2. Therefore, this property of our algorithm enables, for instance, a fast algorithm for computing APSP in the Congested Clique model. We emphasize that, unlike the matrix multiplication algorithms of [7], we are not aware of a similar sparse matrix multiplication algorithm existing in the literature of parallel settings.

Furthermore, we leverage our techniques to obtain a deterministic algorithm for sparsity-aware triangle listing in the Congested Clique model, in which each triangle needs to be known to some node. This problem has been tackled (implicitly) in the Congested Clique model for the first time by Dolev et al. [10], providing two deterministic algorithms. Later, [23, 16] showed a $\tilde{\Omega}(n^{1/3})$ lower bound in general graphs. Pandurangan et al. [23] showed a *randomized* triangle listing algorithm, with the same round complexity as we obtain.

1.1 Our contribution

For a matrix A, let $nz(A)$ be its number of nonzero elements. Our main contribution is an algorithm called SMM (Sparse Matrix Multiplication), for which we prove the following.

▶ **Theorem 1.** *Given two $n \times n$ matrices S and T, Algorithm SMM deterministically computes the product $P = S \cdot T$ over a semiring in the* Congested Clique *model, completing in $O(nz(S)^{1/3} nz(T)^{1/3}/n + 1)$ rounds.*[1]

An important case of Theorem 1, especially when squaring the adjacency matrix of a graph in order to solve graph problems, is when the sparsities of the input matrices are roughly the same. In such a case, Theorem 1 gives the following.

▶ **Corollary 2.** *Given two $n \times n$ matrices S and T, where $O(nz(S)) = O(nz(T)) = m$, Algorithm SMM deterministically computes the product $P = S \cdot T$ over a semiring in the* Congested Clique *model, within $O(m^{2/3}/n + 1)$ rounds.*

Notice that for $m = O(n^2)$, Corollary 2 gives the same complexity of $O(n^{1/3})$ rounds as given by the semiring multiplication of [7].

We apply Algorithm SMM to 4-cycle counting, obtaining the following.

▶ **Theorem 3.** *There is a deterministic algorithm that computes the number of 4-cycles in an n-node graph G in $O(m^{2/3}/n + 1)$ rounds in the* Congested Clique *model, where m is the number of edges of G.*

Notice that for $m = O(n^{3/2})$ this establishes 4-cycle counting in a constant number of rounds.

As described earlier, our algorithm is fast also in the case where only one of the input matrices is sparse, as stated in the following corollary of Theorem 1.

▶ **Corollary 4.** *Given two $n \times n$ matrices S and T, where $\min\{O(nz(S)), O(nz(T))\} = m$, Algorithm SMM deterministically computes the product $P = S \cdot T$ over a semiring in the* Congested Clique *model, within $O((m/n)^{1/3} + 1)$ rounds.*

This allows us to compute powers that are larger than 2 of a sparse input matrix. Although we cannot enjoy the guarantees of our algorithm when repeatedly squaring a matrix, because this may require multiplying dense matrices, we can still repeatedly increase its power by 1. This gives the following for computing APSP, whose comparison to the state-of-the-art depends on trade-off between the number of edges in the graph and its diameter.

▶ **Theorem 5.** *There is a deterministic algorithm that computes unweighted undirected APSP in an n-node graph G in $O(D((m/n)^{1/3} + 1))$ rounds in the* Congested Clique *model, where m is the number of edges of G and D is its diameter.*

For comparison, the previously known best complexity of unweighted undirected APSP is $O(n^{1-2/\omega})$, given by[7, 14], which is currently known to be bounded by $O(n^{0.158})$. For a graph with a number of edges that is $m = o(n^{4-6/\omega}/D^3)$, which is currently $o(n^{1.474}/D^3)$, our algorithm improves upon the latter.

Lastly, we leverage the routing techniques developed in our sparse matrix multiplication algorithm in order to introduce an algorithm for the triangle listing problem in the Congested Clique model.

[1] Since we minimize communication rather than element-wise multiplications, the *zero element* does not have to be the zero element of the semiring - any *single* element may be chosen to not be explicitly communicated.

▶ **Theorem 6.** *There is a deterministic algorithm for triangle listing in an n-node, m-edge graph G in $O(m/n^{5/3}+1)$ rounds in the* Congested Clique *model.*

For comparison, two deterministic algorithms by Dolev et al. [10] take $\tilde{O}(n^{1/3})$ and $O(\lceil \Delta^2/n \rceil)$ rounds, while the sparsity-aware randomized algorithm of Pandurangan et al. [23] completes in $\tilde{O}(m/n^{5/3})$, w.h.p. Notice that for general graphs, our algorithm matches the lower bound of $\tilde{\Omega}(n^{1/3})$ by[23, 16]. Additionally, our algorithm for triangle listing implies a triangle counting algorithm. A triangle counting algorithm whose complexity depends on the arboricity A of the graph is given in [10]. Their algorithm completes in $O(A^2/n+\log_{2+n/A^2} n)$ rounds. Since $A \geq m/n$, this gives a complexity of $\Omega(m^2/n^3)$, upon which our algorithm provides more than a cubic improvement. The previously known best complexity of triangle and 4-cycle counting in general graphs is $O(n^{1-2/\omega})$, given by[7], which is currently known to be bounded by $O(n^{0.158})$. For a graph with a number of edges that is $m = o(n^{8/3-2/\omega})$, which is currently $o(n^{1.824})$, our algorithm improves upon the latter.

The sections containing *triangle listing*, *APSP*, and *4-cycle counting* are in the full paper [8].

1.2 Challenges and Our Techniques

Given two $n \times n$ matrices S and T, denote their product by $P = S \cdot T$, for which $P[i][j] = \sum_{k=1}^{n} S[i][k]T[k][j]$. A common way of illustrating the multiplication is by a *3-dimensional cube* of size $n \times n \times n$, in which the entry (i,j,k) corresponds to the element-wise product $S[i][k]T[k][j]$. In other words, two dimensions of the cube correspond to the matrices S and T, and the third dimension corresponds to element-wise products. Each index of the third dimension is a *page*, and P corresponds to the element-wise summation of all n pages.

In essence, the task of distributed matrix multiplication is to assign each of the n^3 element-wise multiplications to the nodes of the network, in a way which minimizes the amount of communication that is required.[2] This motivates the goal of assigning the element-wise products to the nodes in a way that balances the number of non-zero elements in S and T that need to be communicated among the nodes, as this is the key ingredient towards minimizing the number of communication rounds. The main obstacle is that a sparse input matrix may be unbalanced, leading to the existence of nodes whose element-wise multiplication operation assignment requires them to obtain many nonzero elements of the input matrices that originally reside in other nodes, and thus necessitating much communication.

As we elaborate upon in Section 1.3, algorithms for the parallel settings, which encounter the same hurdle, typically first permute the rows and columns of the input matrices in an attempt to balance the structure of the non-zero entries. Ballard et al. [3] write: "While a priori knowledge of sparsity structure can certainly reduce communication for many important classes of inputs, we are not aware of any algorithms that dynamically determine and efficiently exploit the structure of general input matrices. In fact, a common technique of current library implementations is to randomly permute rows and columns of the input matrices in an attempt to destroy their structure and improve computational load balance."

Our high-level approach, which is *deterministic*, is threefold. The first ingredient is splitting the $n \times n \times n$ cube into n equally sized sub-cubes whose dimensions are determined dynamically, based on the sparsity of the input matrices. The second is indeed permuting the input matrices S and T into two matrices S' and T', respectively. We do so in a subtle

---

[2] We consider all n^3 element-wise multiplications rather than Strassen-like algorithms since we work over a semiring and not a ring.

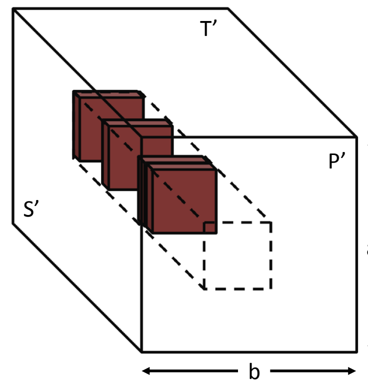


Figure 1 An illustration of the multiplication cube for $P' = S'T'$. Each sub-matrix is assigned to n/ab nodes, with a not necessarily consecutive page assignment that is computed on-the-fly to minimize communication.

manner, for which the resulting matrices exhibit some nice balancing property.[3] The third ingredient is the innovative part of our algorithm, which assigns the computation of pages of different sub-matrices across the nodes in a *non-consecutive* manner. We elaborate below about these key ingredients, with the aid of Figure 1.

Permuting the input matrices: We employ standard parallelization of the task of computing the product matrix P, by partitioning P into ab equal sized $n/a \times n/b$ sub-matrices denoted by $P_{i,j}$ for $i \in [a], j \in [b]$, and assigning n/ab nodes for computing each sub-matrix.

To this end, we leverage the simple observation that the multiplication of permutations of the rows of S and the columns of T results in a permutation of the product of S and T, which can be easily inverted. This observation underlies the first part of our algorithm, in which the nodes permute the input matrices, such that the number of non-zero entries from S and T that are required for computing each $n/a \times n/b$ sub-matrix are roughly the same across the $a \cdot b$ sub-matrices. We call the two matrices, S' and T', that result from the permutations, *sparsity-balanced matrices with respect to* (a, b). The rest of our algorithm deals with computing the product of two such matrices. This part inherently includes a computation of the best choice for a and b for minimizing the communication.

Assigning pages to nodes: To obtain each sub-matrix $P_{i,j}$, there are n sub-pages $P_{i,j,\ell}$ which need to be computed and summed. For each $P_{i,j}$, this task is assigned to distinct n/ab nodes, each of which computes some of the n sub-pages $P_{i,j,\ell}$ and sums them locally. The local sums are then aggregated and summed, for obtaining $P_{i,j}$. We utilize the commutativity and associativity properties of summation over the semiring in order to assign sub-pages to nodes in a *non-consecutive* manner, such that the nodes require receiving a roughly equal number of non-zero entries in order to compute their assigned sub-pages.

[3] Note, we do not assume that balancing the distribution of non-zero elements gives a balanced local computation. Our balancing is done for the amount of communication: we assign the amount and identity of matrix entries that should be sent and received by each node in a way that will balance the communication, not necessarily the local computation.

Assigning non-zero matrix entries to nodes: For fast communication in the Congested Clique model using Lenzen's routing scheme (see Section 1.4), it is moreover paramount that the nodes also *send* a roughly equal amount of non-zero matrix entries. However, it may be the case that a certain row, held by a node v, contains a significantly larger number of non-zero entries as compared with other rows. Therefore, we rearrange the entries held by each node such that every node holds a roughly equal amount of non-zero entries that need to be sent to other nodes for computing the n^3 products. Notice that in this step we do not rearrange the rows or columns of S or T, rather, we redistribute the entries of S and T. Thus, a node may hold values which originate from different rows.

Routing non-zero elements: Crucially, the assignments made above, for addressing the need to balance sending and receiving, are not global knowledge. That is, for every $P_{i,j}$, the corresponding n/ab nodes decide which matrix entries are received by which node, but this is unknown to the other nodes, who need to send this information. Likewise, the redistribution of entries of S and T across the nodes is not known to all nodes. Nonetheless, clearly, a node must know the destination of each message it needs to send. As a consequence, we ultimately face the challenge of communicating some of this local knowledge. In our solution, a node that needs to receive information from a certain column of S (or row of T) sends a request to the nodes holding subsequences of that column (or row) without knowing the exact partition into subsequences. The nodes then deliver the non-zero entries of this column (or row), which allow computing the required element-wise multiplications.
Our solutions to the three challenges described above, for *sending* and *receiving* as small as possible amounts of information and for resolving a corresponding *routing*, and their combination, are the main innovation of our algorithm.

1.3 Related work

Matrix multiplication in the Congested Clique model: A randomized Boolean matrix multiplication algorithm was given by Drucker et al. [11], completing in $O(n^{\omega-2})$ rounds, where ω is the exponent of sequential matrix multiplication. The best currently known upper bound is $\omega < 2.3728639$ [13], implying $O(n^{0.372})$ rounds for the above.

Later, Censor-Hillel et al. [7] gave a deterministic algorithm for (general) matrix multiplication over semirings, completing in $O(n^{1/3})$ rounds, and a deterministic algorithm for (general) matrix multiplication over rings, completing in $O(n^{1-2/\omega})$ rounds, which by the current known upper bound on ω is $O(n^{0.158})$. The latter is a Strassen-like algorithm, exploiting known schemes for computing the product of two matrices over a ring without directly computing all n^3 element-wise multiplications. Then, Le Gall [14] provided fast algorithms for multiplying rectangular matrices and algorithms for computing multiple instances of products of independent matrices.

Related graph computations in the Congested Clique model: Triangle counting in the Congested Clique model was addressed by Dolev et al. [10], who provided a deterministic $\tilde{O}(n^{d-2/d})$-round algorithm for counting the number of appearances of any d-node subgraph, giving triangle counting in $\tilde{O}(n^{1/3})$ rounds. To speed up the computation for sparse instances, [10] show that every node in a graph with a maximum degree of Δ can learn its 2-hop neighborhood within $O(\Delta^2/n)$ rounds, implying the same round complexity for triangle counting. They also showed a deterministic triangle counting algorithm completing in $\tilde{O}(A^2/n+\log_{2+n/A^2} n)$ rounds, where A is the arboricity of the input graph, i.e., the minimal number of forests into which the set of edges can be decomposed. Note that a graph with

arboricity A has at most An edges, but there are graphs with arboricity A and a significantly smaller number of edges. Since it holds that $A \geq m/n$, this implies a complexity of $\Omega(m^2/n^3)$ for their triangle counting algorithm, upon which our $O(m^{2/3}/n+1)$-round algorithm provides a cubic improvement. The deterministic matrix multiplication algorithm over rings of [7] directly gives a triangle counting algorithm with $O(n^{1-2/\omega})$ rounds.

For 4-cycle counting, the algorithm of [10] completes in $\tilde{O}(n^{1/2})$ rounds, and the matrix multiplication algorithm of [7] implies a solution in $O(n^{1-2/\omega})$ rounds.

For APSP, the matrix multiplication algorithms of [7] give $O(n^{1-2/\omega})$ for the unweighted undirected case. For weighted directed APSP, $\tilde{O}(n^{1/3})$ rounds are given in [7], and improved algorithms for weighted (directed and undirected) APSP are given in [14]. We mention that our technique could allow for computing weighted APSP, but the cost would be too large due to our iterative multiplication (as opposed to the previous algorithms that can afford iterative *squaring*). Algorithms for approximations of APSP are given in [22, 7, 14].

Note that for all graph problems, Lenzen's routing scheme [21] (see Section 1.4) implies that every node can learn the entire structure of G within $O(m/n)$ rounds, where m is the number of edges (this can also be obtained by a simpler scheme).

Sequential/Parallel matrix multiplication: Additional related work on sequential and parallel matrix multiplication can be found in the full paper [8].

1.4 Preliminaries

Model: The Congested Clique model consists of a set $[n] = \{1, \ldots, n\}$ of nodes in a fully connected synchronous network, limited by a bandwidth of $O(\log n)$ bits per message.

In an instance of multiplication of two matrices S and T, the input to each node v is row v of each matrix and its output should be row v of $P = S \cdot T$. For a graph problem over a graph G of n nodes, we identify the nodes of the Congested Clique model with the nodes of G, and the input to node v in the Congested Clique model is its input in G.

As defined earlier, for a matrix A we denote by $nz(A)$ the number of non-zero elements of A. Throughout the paper, we also need to refer to the number of non-zero elements in certain sub-matrices or sequences. We will therefore overload this notation, and use $nz(X)$ to denote the number of non-zero elements in any object X.

A pair of integers (a, b) is *n-split* if $a, b \in [n]$, both a and b divide n, and $n/ab \geq 1$. The requirement that a and b divide n is for simplification only and could be omitted. Eventually, the n-split pair that will be chosen is $a = n \cdot nz(S)^{1/3}/nz(T)^{2/3}$ and $b = n \cdot nz(T)^{1/3}/nz(S)^{2/3}$.

For a given n-split pair (a, b), it will be helpful to associate each node $v \in [n]$ with three indices, two indicating the $P_{i,j}$ sub-matrix to which the node is assigned, and one distinguishing it from the other nodes assigned to $P_{i,j}$. Hence, we denote each node v also as $v_{i,j,k}$, where $i \in [a]$, $j \in [b]$, and $k \in [n/ab]$. The assignment of indices to the nodes can be any arbitrary one-to-one function from $[n]$ to $[a] \times [b] \times [n/ab]$.

Throughout our algorithms, we implicitly comply with the following: (I) no information is sent for matrix entries whose value is zero, and (II) when the value of a non-zero entry is sent, it is sent alongside its location in the matrix. Since sending the location within an $n \times n$ matrix requires $O(\log n)$ bits, the overhead in the complexity is constant.

Lenzen's routing scheme: A useful tool in designing algorithms for the Congested Clique model is *Lenzen's routing scheme* [21]. In this scheme, each of the n nodes can send and receive $n-1$ messages (of $O(\log n)$ bits each) in a constant number of rounds. While this is simple to see for the simplest case where each node sends a single message to every other

node, the power of Lenzen's scheme is that it applies to any (multi)set of source-destination pairs, as long as each node is source of at most $n-1$ messages and destination of at most $n-1$ messages. Moreover, the multiset of pairs does not need to be known to all nodes in advance, rather each sender only needs to know the recipient of its messages. Employing this scheme is what underlies our incentive for balancing the number of messages that need to be sent and received by all the nodes.

Useful combinatorial claims: The following are simple combinatorial claims that we use for routing messages in a load-balanced manner.

▶ Claim 1. Let $A=(a_1,\dots,a_t)$ be a finite set and let $1 \le c \le t$ be an integer. There exists a partition of A into $\lceil t/c \rceil$ subsets of size at most $c+1$ each.

▶ Claim 2. Given any n finite sets, $A^i=(a^i_1,\dots,a^i_{t_i})$ for $1 \le i \le n$. Let $avg = (\sum_{1\le i \le n} t_i)/n$. There exists a partition of each A^i into $\lceil t_i/avg \rceil$ subsets of size at most $avg+1$ each, such that the total number of subsets is at most $2n$.

▶ Claim 3. Given a sorted finite multiset $A=(a_1,\dots,a_n)$ of natural numbers, an integer $x \in \mathbb{N}$ such that for all $i \in [n]$ it holds that $a_i \le x$, and an integer k that divides n, there exists a partition $A=\cup_{i=j}^{k} A_j$ into k multisets A_j, $1 \le j \le k$, of equal size n/k, such that for all $1 \le j \le k$ it holds that $sum(A_j) \le sum(A)/k + x$.

2 Fast Sparse Matrix Multiplication

Our main result is Theorem 1, stating the guarantees of our principal algorithm SMM (Sparse Matrix Multiplication) for fast multiplication of sparse matrices. Algorithm SMM first manipulates the structure of its input matrices and then calls algorithm SBMM (Sparse Balanced Matrix Multiplication), which solves the problem of fast sparse matrix multiplication under additional assumptions on the distributions of non-zero elements in the input matrices, which are defined next. In Section 2.1, we show how SMM computes general matrix multiplication $P=ST$, given Algorithm SBMM and Theorem 8. Algorithm SBMM, and Theorem 8 which states its guarantees, are deferred to Section 2.2.

Theorem 1 (restated). *Given two $n \times n$ matrices S and T, Algorithm SMM deterministically computes the product $P=S \cdot T$ over a semiring in the* Congested Clique *model, completing in $O(nz(S)^{1/3} nz(T)^{1/3}/n+1)$ rounds.*

We proceed to presenting Theorem 8 which discusses SBMM. SBMM multiplies matrices S' and T' in which the non-zero elements are roughly balanced between portions of the rows of S' and columns of T'. In what follows, for a matrix A, the notation $A[x:y][*]$ refers to rows x through y of A and the notation $A[*][x:y]$ refers to columns x through y of A. In the following definition we capture the needed properties of well-balanced matrices.

▶ **Definition 7.** Let S and T be $n \times n$ matrices and let (a,b) be an n-split pair. For every $i \in [a]$ and $j \in [b]$, denote $S_i = S[(i-1)(n/a)+1 : i(n/a)][*]$ and $T_j = T[*][(j-1)(n/b)+1 : j(n/b)]$. We say that S and T are a *sparsity-balanced pair of matrices with respect to* (a,b), if:

- S-condition: For every $i \in [a]$, $nz(S_i) \le nz(S)/a + n$.
- T-condition: For every $j \in [b]$, $nz(T_j) \le nz(T)/b + n$.

These conditions ensure that *bands* of adjacent rows of S and columns of T contain roughly the same number of non-zero elements. We can now state our theorem for multiplying sparsity-balanced matrices, which summarizes our algorithm SBMM.

▶ **Theorem 8.** *Given two $n \times n$ matrices S and T and an n-split pair (a, b), if S and T are a sparsity-balanced pair with respect to (a, b), then Algorithm SBMM deterministically computes the product $P = S \cdot T$ over a semiring in the* Congested Clique *model, completing in $O(nz(S) \cdot b/n^2 + nz(T) \cdot a/n^2 + n/ab)$ rounds.*

We show that $O(1)$ rounds are sufficient in the Congested Clique for transforming any two general matrices S and T to sparsity-balanced matrices S' and T' by invoking standard matrix permutation operations. Therefore, in essence, Algorithm SMM performs permutation operations on S and T, generating the matrices S' and T', respectively, invokes SBMM on S' and T' to compute $P' = S'T'$, and finally recovers P from P'.

2.1 Fast General Sparse Matrix Multiplication - Algorithm SMM

Algorithm Description: First, each node distributes the entries in its row of T to other nodes in order for each node to obtain its column in T. Then, the nodes broadcast the number of non-zero elements in their respective row of S and column of T, in order for all nodes to compute $nz(S)$ and $nz(T)$. Having this information, the nodes locally compute the n-split pair (a, b) that minimizes the expression $nz(S) \cdot b/n^2 + nz(T) \cdot a/n^2 + n/ab$, which describes the round complexities of each of the three parts of Algorithm SBMM. It can be shown that the pair $(n \cdot nz(S)^{1/3}/nz(T)^{2/3}, n \cdot nz(T)^{1/3}/nz(S)^{2/3})$ minimizes this expression. Then, the nodes permute the rows of S and columns of T so as to produce matrices S' and T' which have the required balance. Subsequently, Algorithm SBMM is executed on the permuted matrices S' and T', followed by invoking the inverse permutations on the product $P' = S'T'$ in order to obtain the product $P = S \cdot T$ of the original matrices. A pseudocode of SMM is given in Algorithm 1.

Proof of Theorem 1. To prove correctness, we need to show that the matrices S' and T' computed in Line 10 are a sparsity-balanced pair of matrices with respect to the n-split pair (a, b) that is determined in Line 8. Once this is proven, the correctness of the algorithm is as follows. In Lines 1-12 the matrices S' and T' are computed and are distributed among the nodes such that each node $v \in [n]$ holds row v of S and column v of T. The loop of Line 13 is only for consistency, having the input to SBMM be the respective rows of both S' and T'. Assuming the correctness of algorithm SBMM given in Theorem 8, the matrix P' computed in Line 15 is the product $P' = S'T'$. Finally, in the last loop, node v receives row v of $P = A_\sigma^{-1} P' A_\tau^{-1}$, completing the correctness of the Algorithm SMM.

We now show that S' and T' are indeed a sparsity-balanced pair of matrices with respect to (a, b). To this end, we first need to show that for all $i \in [a]$, the number of non-zero elements in S'_i is at most $nz(S')/a + n$. By construction, the number of non-zero elements in $S'_i = S'[(i-1)(n/a) + 1 : i(n/a)][*]$ is exactly $sum(A_i^S)$ of the partition computed in Line 9. By Claim 3 this is bounded by $sum(A)/k + x$, which in our case is $nz(S)/a + n = nz(S')/a + n$. Thus, S' satisfies the S-condition of Definition 7. A similar argument shows that T' satisfies the T-condition of Definition 7.

For the complexity, we sum the number of rounds as follows. The first loop allows every node v to obtain column v of T, while in the second loop the nodes exchange the sums of non-zero elements in rows and columns of S and T, respectively. Even without the need to resort to Lenzen's routing scheme, both of these loops can be completed within $O(1)$ rounds. A similar argument shows that $O(1)$ rounds suffice for permuting S and T into S' and T', and for permuting P' back into P. Thus, all lines of the pseudocode excluding Line 15 complete in $O(1)$ rounds. This implies that the complexity of Algorithm SMM equals that of Algorithm SBMM when given S', T', a, and b as input. By Theorem 8 and due to the choice of a and

Algorithm 1: SMM (S, T): Computing the product $P = S \cdot T$. Code for node $v \in \{1, \dots, n\}$.

```
1  foreach u ∈ [n], u ≠ v do
2      send T[v][u] to node u
3  foreach u ∈ [n], u ≠ v do
4      send nz(S[v][∗]) to node u
5      send nz(T[∗][v]) to node u
6  nz(S) ← Σ_{u∈[n]} nz(S[u][∗])
7  nz(T) ← Σ_{u∈[n]} nz(T[∗][u])
8  (a, b) ← argmin_{n-split pairs (a,b)} {nz(S)·b/n^2 + nz(T)·a/n^2 + n/ab}
9  Let A^S_1, ..., A^S_a be the partition of the sorted multiset of {nz(S[u][∗])|u ∈ [n]} into a
     multisets, and A^T_1, ..., A^T_b be the partition of the sorted multiset of
     {nz(T[∗][u])|u ∈ [n]} into b multisets, both proven to exist in Claim 3 (with x = n).
10 Let σ be a permutation for which its n × n permutation matrix A_σ is such that the
     rows of the matrix S' = A_σ S that correspond to any single A^S_u are adjacent, and let
     τ be a permutation for which its n × n permutation matrix A_τ is such that the
     columns of the matrix T' = T A_τ that correspond to any single A^T_u are adjacent.
11 send S[v][∗] to node σ(v)
12 send T[∗][v] to node τ(v)
13 foreach u ∈ [n], u ≠ v do
14     send T'[u][v] to node u
15 P' ← SBMM(S', T', a, b)
16 foreach u ∈ [n], u ≠ v do
17     send P'[σ^{-1}(v)][τ^{-1}(u)] to node u
```

b in Line 8, this complexity is $O(\min_{n\text{-split pairs }(a,b)}\{nz(S) \cdot b/n^2 + nz(T) \cdot a/n^2 + n/ab\})$. Choosing $a = n \cdot nz(S)^{1/3}/nz(T)^{2/3}$ and $b = n \cdot nz(T)^{1/3}/nz(S)^{2/3}$ gives a complexity of $O(nz(S)^{1/3}nz(T)^{1/3}/n + 1)$ rounds, which can be shown to be optimal. ◀

2.2 Fast Sparse Balanced Matrix Multiplication - Algorithm SBMM

Here we present SBMM and prove Theorem 8. We begin with a short overview of the algebraic computations and node allocation in SBMM. We then proceed to presenting a communication scheme detailing how to perform the computations of SBMM in the Congested Clique model in $O(M_S \cdot b/n^2 + M_T \cdot a/n^2 + n/ab)$ rounds of communication.

Algorithm Description: Consider the partition of P into ab rectangles, such that $\forall (i,j) \in [a] \times [b]$, *sub-matrix* $P_{i,j} = P[(i-1)(n/a)+1 : i(n/a)][(j-1)(n/b)+1 : j(n/b)]$. Each sub-matrix $P_{i,j}$ is an $n/a \times n/b$ matrix, i.e., has n^2/ab entries. Notice that $P_{i,j} = S_i \cdot T_j$. We assign the computation of $P_{i,j}$ to a unique set of n/ab nodes $N_{i,j} = \{v_{i,j,k} | k \in [n/ab]\}$.

In the initial phase of algorithm SBMM, for every $(i,j) \in [a] \times [b]$, each non-zero element of S_i and T_j is sent to some node in $N_{i,j}$. Due to the sparsity-balanced property of S and T, all S_i's have roughly the same amount of non-zero elements, and likewise all T_j's. Therefore, each set of nodes $N_{i,j}$ receives roughly the same amount of non-zero elements from S and T.

Within each $N_{i,j}$, the computation of $P_{i,j}$ is carried out according to the following framework. For $\ell \in [n]$, denote each page of $P_{i,j}$ by $P_{i,j,\ell} = S_i[*][\ell] \cdot T_j[\ell][*]$. The computation

Algorithm 2: SBMM (S,T,a,b): Computing the product $P = ST$, for S and T that are sparsity-balanced w.r.t. (a, b). Code for node $v_{i,j,k}$.

1 ExchangeInfo (S, T, a, b)
2 Locally compute $P_{i,j,\ell}$ for every $\ell \in A_{i,j,k}$
3 Locally compute $P^k_{i,j} = \sum_{\ell \in A_{i,j,k}} P_{i,j,\ell}$
4 **foreach** $t \in [n/a]$ **do**
5   **send** $P^k_{i,j}[t][*]$ to node of respective row
6 **foreach** $\ell \in [n]$ **do**
7     $P[v][\ell] \leftarrow$ sum of n/ab respective elements received for this entry

of the n different $P_{i,j,\ell}$ sub-matrices is split among the nodes in $N_{i,j}$ as follows: The set $[n]$ is partitioned into $A_{i,j,1}, \ldots, A_{i,j,n/ab}$ such that for each $k \in [n/ab]$, node $v_{i,j,k} \in N_{i,j}$ is required to compute the entries of the matrices in the set $\{P_{i,j,\ell} | \ell \in A_{i,j,k}\}$. Then, node $v_{i,j,k} \in N_{i,j}$ locally sums its computed sub-matrices to produce $P^k_{i,j} = \sum_{\ell \in A_{i,j,k}} P_{i,j,\ell}$. Clearly, due to the associativity and commutativity of the addition operation in the semiring, it holds that $P_{i,j} = \sum_{\ell \in [n]} P_{i,j,\ell} = \sum_{k \in [n/ab]} P^k_{i,j}$. Therefore, once every node $v_{i,j,k}$ has $P^k_{i,j}$, the nodes can collectively compute P, and redistribute its entries in a straightforward manner such that each node obtains a distinct row of P.

Implementing SBMM: A pseudocode for Algorithm SBMM is given in Algorithm 2, which consists of three components: exchanging information between the nodes such that every node $v_{i,j,k}$ has the required information for computing $P_{i,j,\ell}$ for every $\ell \in A_{i,j,k}$, local computation of $P^k_{i,j}$ for each $(i, j, k) \in [a] \times [b] \times [n/ab]$ and, finally, the communication of the $P^k_{i,j}$ matrices and assembling of the rows of P.

The technical challenge is in Line 1, upon which we elaborate below. In Lines 2-3, only local computations are performed, resulting in each node $v_{i,j,k}$ holding $P^k_{i,j}$. In Line 4, each node sends each row of its sub-matrix $P^k_{i,j}$ to the appropriate node, so that in Line 6 each node can sum this information to produce its row in P. In the full version [8], we prove Lemma 9.

▶ **Lemma 9.** *Lines 2-6 of Algorithm 2 complete in $O(n/ab)$ rounds, producing a row of $P = ST$ for every node.*

The remainder of this section is dedicated to presenting and analyzing Line 1. During this part of the algorithm, for every $(i, j) \in [a] \times [b]$, each entry in S_i and T_j needs to be sent to a node in $N_{i,j}$. As per our motivation throughout the entire algorithm, we strive to achieve this goal in a way which ensures that all nodes send and receive roughly the same number of messages. This leads to the following three challenges which we need to overcome.

Sending Challenge: Initially, node v holds row v of S and row v of T. Every column v of S needs to be sent to b nodes - one node in each $N_{i,j}$ for an appropriate $i \in [a]$ and every $j \in [b]$. Similarly, every row of T needs to be sent to a nodes - one in each $N_{i,j}$ for an appropriate $j \in [b]$ and every $i \in [a]$. If we were to trivially choose node v to send all these messages, then node v would need to send $nz(S[*][v]) \cdot b + nz(T[v][*]) \cdot a$ messages. Since $nz(S[*][v])$ and $nz(T[v][*])$ may widely vary for different values of v, it may be the case that some nodes send a significant amount of messages while others are relatively silent.

Algorithm 3: ExchangeInfo (S,T,a,b): Sending each entry of S_i, T_j to a node in $N_{i,j}$, for every $(i,j) \in [a] \times [b]$.

1 Compute-Sending
2 Compute-Receiving
3 Resolve-Routing

Receiving Challenge: Since S and T are sparsity-balanced w.r.t. (a,b), for every $(i,j) \in [a] \times [b]$ it holds that the number of messages to be received by each set of nodes $N_{i,j}$ is at most $nz(S)/a + nz(T)/b + 2n$. This ensures that each node set $N_{i,j}$ receives roughly the same amount of messages as every other node set. The challenge remains to ensure that *within* any given node set $N_{i,j}$, every node receives roughly the same number of messages.

Routing Challenge: When overcoming the above mentioned challenges in a non-trivial manner, all nodes locally determine that they are senders and recipients of certain messages with the guarantee that each node sends and receives roughly the same number of messages. However, these partitions of sending and receiving messages are obtained independently and thus are not global knowledge; a sender of a message *does not necessarily know* who the recipient is. The routing challenge is thus to ensure that each node associates the correct recipient with every message that it sends.

2.2.1 ExchangeInfo (S,T,a,b)

We next present our implementation of ExchangeInfo (S,T,a,b) which solves the above challenges in an on-the-fly manner. To simplify the presentation, we split ExchangeInfo (S,T,a,b) into its three components, as given in the pseudocode of Algorithm 3.

Compute-Sending: In Compute-Sending, whose pseudocode is given in Algorithm 4, we overcome the sending challenge. The nodes communicate the distribution of non-zero elements across the columns of S and the rows of T and reorganize the entries held by each node such that all nodes hold roughly the same amount of non-zero elements of S and T.

Notably, in order to enable fast communication in Resolve-Routing, Algorithm 4 must guarantee no node holds entries of more than two columns of S and two rows of T.

▶ **Lemma 10.** *Algorithm 4 completes in $O(1)$ rounds, after which the entries of S and T are evenly redistributed across the nodes such that every node holds elements from at most 2 columns of S and 2 rows of T and such that every node v knows for every node u the indices of the two columns of S and two rows of T from which the elements which u holds are taken.*

Proof. In Lines 2, 4, 5 the nodes exchange entries of S such that each node holds a distinct column of S, and knows the number of non-zero entries in each column of S and in each row of T. This allows local computation of the average number of non-zeros in the following two lines, as well as locally computing the (same) partition into subsequences.

By Claim 2, in total across all n columns there are at most $2n$ subsequences of entries from S, and similarly there are at most $2n$ subsequences from T. Since $\forall u \in [n]$, all nodes know $nz(S[*][u])$ and $nz(T[u][*])$, then all nodes know how many subsequences are created for each u. Thus, all nodes can agree in Line 9 on the assignment of the subsequences, with each node assigned at most 2 subsequences of entries of S and 2 of entries of T. Crucially for what follows, all the nodes know the column ℓ in S or the row ℓ in T to which the

Algorithm 4: Compute-Sending: Code for node $v_{i,j,k}$.

1 **foreach** $u \in [n]$, $u \neq v$ **do**
2   **send** $S[u][v]$ to node u
3 **foreach** $u \in [n]$, $u \neq v$ **do**
4   **send** $nz(S[*][v])$ to node u
5   **send** $nz(T[v][*])$ to node u
6 $avg(S) \leftarrow (\sum_{u\in[n]} nz(S[*][u]))/n$
7 $avg(T) \leftarrow (\sum_{u\in[n]} nz(T[u][*]))/n$
8 Let $S_1^v, \ldots, S^v_{\lceil nz(S[*][v])/avg(S)\rceil}$ be a partition of the non-zero elements of $S[*][v]$ into sets of size at most $avg(S)+1$ and let $T_1^v, \ldots, T^v_{\lceil nz(T[v][*])/avg(T)\rceil}$ be a partition of the non-zero elements of $T[v][*]$ into sets of size at most $avg(T)+1$, both proven to exist in Claim 1. We refer to these sets as *subsequences*.
9 Assign two subsequences of S, denote by $B_S(v)$, and two subsequences of T, denote by $B_T(v)$ to each node v. For each subsequence B, denote by $v(B)$ the node to which B is assigned.
10 **foreach** $B \in \{S_1^v, \ldots, S^v_{\lceil nz(S[*][v])/avg(S)\rceil}, T_1^v, \ldots, T^v_{\lceil nz(T[v][*])/avg(T)\rceil}\}$ **do**
11   **send** B to node $v(B)$

subsequence B belongs. We denote this index $\ell(B)$. The entries of each subsequence B are then sent to its node $v(B)$ in the following loop.

For the round complexity, note that a node v sends a single message to every other node in each of Lines 2, 4, and 5. The rest of the computation until Line 9 is done locally. Therefore, these lines complete within 3 rounds.

In the last loop of Algorithm 4, node v potentially sends all subsequences with entries from column v of S and row v of T. Due to the facts that each subsequence is sent only once, no subsequences overlap, and all the subsequences which v send are parts of a single column of S and a single row of T, node v sends at most $2n$ messages during this loop. Additionally, since every node receives at most 4 subsequences and each subsequence consists of most n entries, each node receives at most $4n$ messages. Thus, by using Lenzen's routing scheme, this completes in $O(1)$ rounds as well. ◀

Compute-Receiving: The pseudocode for Compute-Receiving is given in Algorithm 5. This algorithm assigns the $P_{i,j,\ell}$ matrices to different nodes in $N_{i,j}$. Specifically, each node $v_{i,j,k}$ in $N_{i,j}$ is assigned ab such matrices, while verifying that all nodes in $N_{i,j}$ require roughly the same amount of non-zero entries from S and T in order to compute all their assigned $P_{i,j,\ell}$ matrices. Since each sub-matrix $P_{i,j,\ell}$ is defined as $P_{i,j,\ell} = S_i[*][\ell] \cdot T_j[\ell][*]$, we define the communication cost of computing $P_{i,j,\ell}$ to be $w(P_{i,j,\ell}) = nz(S_i[*][\ell]) + nz(T_j[\ell][*])$. By this definition, in order to obtain that each node in $N_{i,j}$ requires roughly the same amount of messages in order to compute all of its assigned $P_{i,j,\ell}$, we assign the $P_{i,j,\ell}$ matrices to the nodes of $N_{i,j}$ such that the total communication cost, as measured by w, of all matrices assigned to a given node is roughly the same for all nodes.

▶ **Lemma 11.** *Algorithm 5 completes in $O(1)$ rounds, after which each node $v_{i,j,k}$ is assigned a subset $A_{i,j,k} \subseteq [n]$, s.t. $\forall k \in [n/ab]$ it holds that $\sum_{\ell\in A_{i,j,k}} w(P_{i,j,\ell}) \leq \frac{n}{ab}\sum_{\ell\in[n]} w(P_{i,j,\ell}) + 2n$.*

Proof. The loop of Line 1 provides each node of $N_{i',j'}$ with the number of non-zero elements in each column of $S_{i'}$ and each row of $T_{j'}$. This allows the nodes to compute the required

Algorithm 5: Compute-Receiving: Code for node $v_{i,j,k}$.

```
1  foreach N_{i',j'}, i', j' ∈ [a] × [b] do
2      foreach u ∈ N_{i',j'} do
3          foreach B ∈ B_S(v) do
4              send nz(S_{i'} ∩ B) to node u
5          foreach B ∈ B_T(v) do
6              send nz(T_{j'} ∩ B) to node u
7  foreach ℓ ∈ [n] do
8      w(P_{i,j,ℓ}) ← nz(S_i[*][ℓ]) + nz(T_j[ℓ][*])
9  Let A'_{i,j,1}, ..., A'_{i,j,n/ab} be a partition of the sorted multiset {w(P_{i,j,ℓ})|ℓ ∈ [n]} into
   n/ab multisets with a bound x = 2n on its elements, proven to exist in Claim 3.
10 Let A_{i,j,1}, ..., A_{i,j,n/ab} be a partition of [n] such that for every k ∈ [n/ab],
   A'_{i,j,k} = {w(P_{i,j,ℓ})|ℓ ∈ A_{i,j,k}}. .
```

communication costs in Line 7. Claim 3 implies that after executing Line 10, each node $v_{i,j,k}$ is assigned a subset $A_{i,j,k} \subseteq [n]$, such that for every $k \in [n/ab]$ it holds that $\sum_{\ell \in A_{i,j,k}} w(P_{i,j,\ell}) \leq \frac{n}{ab} \sum_{\ell \in [n]} w(P_{i,j,\ell}) + 2n$.

Every node sends every other node exactly 4 messages throughout the loop in Line 1, while the remaining lines are executed locally for each node, without communication. As such, this completes in $O(1)$ rounds in total. ◀

Resolve-Routing: Roughly speaking, we solve this challenge by having the recipient of each possibly non-zero entry deduce which node is the sender of this entry, and inform the sender that it is its recipient, as follows. At the end of the execution of Compute-Sending in Algorithm 4, every node v has at most two subsequences in $B_S(v)$ and at most two subsequences in $B_T(v)$. Moreover, the subsequence assignment is known to all nodes due to performing the same local computation in Line 9. On the other hand, upon completion of Algorithm 5, node $v_{i,j,k}$ is assigned the task of computing $P_{i,j,\ell}$ for every $\ell \in A_{i,j,k}$. For this, it suffices for $v_{i,j,k}$ to know the non-zero entries of column ℓ of S_i and of row ℓ of T_j.

Hence, in Resolve-Routing, given in Algorithm 6, node $v_{i,j,k}$ sends every index $\ell \in A_{i,j,k}$ to the nodes that hold subsequences of column ℓ in S and row ℓ in T. Notice that $v_{i,j,k}$ does not know which indices inside these columns and rows are non-zero. However, the nodes which hold these subsequences have this information, and respond with the non-zero entries of the respective columns and rows that are part of S_i or T_j. The proof of the following Lemma 12 appears in the full version [8].

▶ **Lemma 12.** *Algorithm 6 completes in $O(nz(S) \cdot b/n^2 + nz(T) \cdot a/n^2 + 1)$ rounds, after which each node $v_{i,j,k}$ has $S[(i-1)(n/a)+1 : i(n/a)][\ell]$ and $T[\ell][(j-1)(n/b)+1 : j(n/b)]$, for every $\ell \in A_{i,j,k}$.*

Proof of Theorem 8. Lemma 12 implies that each node $v_{i,j,k}$ has the required entries of S and T in order to compute $P_{i,j,\ell}$ for every $\ell \in A_{i,j,k}$. Lemma 9 then gives that Algorithm SBMM correctly produces a row of $P = ST$ for each node.

Lemmas 10 and 11 show that Compute-Sending and Compute-Receiving complete in $O(1)$ rounds. Lemma 12 gives the claimed round complexity of $O(nz(S) \cdot b/n^2 + nz(T) \cdot a/n^2 + 1)$

Algorithm 6: Resolve-Routing: Code for node $v_{i,j,k}$.

```
1  foreach ℓ ∈ A_{i,j,k} do
2      foreach node u for which there exists B ∈ B_S(u) such that ℓ(B) = ℓ do
3          send ℓ to node u
4      foreach node u for which there exists B ∈ B_T(u) such that ℓ(B) = ℓ do
5          send ℓ to node u
6  foreach message ℓ received from node v_{i',j',k'} in Line 3 do
7      foreach B ∈ B_S(v) do
8          send S[(i' − 1)(n/a) + 1 : i'(n/a)][ℓ] ∩ B to node v_{i',j',k'}
9  foreach message ℓ received from node v_{i',j',k'} in Line 5 do
10     foreach B ∈ B_T(v) do
11         send T[ℓ][(j' − 1)(n/b) + 1 : j'(n/b)] ∩ B to node v_{i',j',k'}
```

for Resolve-Routing, giving the same total number of rounds for ExchangeInfo. By Lemma 9, the remainder of Algorithm SBMM completes in $O(n/ab)$ rounds, completing the proof. ◀

3 Discussion

This work significantly improves upon the round complexity of multiplying two matrices in the distributed Congested Clique model, for input matrices which are sparse. As mentioned, we are unaware of a similar algorithmic technique being utilized in the literature of parallel computing, which suggests that our approach may be of interest in a more general setting. The central ensuing open question left for future reserach is whether the round complexity of sparse matrix multiplication in the Congested Clique can be further improved.

Finally, an intriguing question is the complexity of various problems in the more general k-machine model [18, 23], where the size of the computation clique is $k << n$. The way of partitioning the data to the nodes is of importance. One may assume that the input to each node consists of n/k unique consecutive rows of S and T, and its output should be the corresponding n/k rows of the product $P = S \cdot T$. Applying our algorithm in this setting gives a round complexity of $O(\min_{n\text{-split pairs }(a,b)} n^2/k^2 + nz(S) \cdot b/k^2 + nz(T) \cdot a/k^2 + n^2/kab + 1)$ rounds, which is $O(n^{2/3} \cdot nz(S)^{1/3} nz(T)^{1/3}/k^{5/3} + 1)$ rounds with the assignment $a = n^{2/3}k^{1/3} \cdot nz(S)^{1/3}/nz(T)^{2/3}$ and $b = n^{2/3}k^{1/3} \cdot nz(T)^{1/3}/nz(S)^{2/3}$. To see why, consider each node as simulating the behavior of n/k *virtual* nodes of the Congested Clique model that belong to the same $N_{i,j}$ set. The round complexity of all steps of the algorithm grows by a multiplicative factor of n^2/k^2, apart from the steps in Algorithm 2 which grow only by a multiplicative factor of n/k, since part of the simulated communication consists of messages sent between virtual nodes that are simulated by the same actual node, and as such do not require actual communication. We ask whether this complexity can be improved for $k << n$.

References

1 Rasmus Resen Amossen and Rasmus Pagh. Faster join-projects and sparse matrix multiplications. In *The 12th International Conference on Database Theory (ICDT)*, pages 121–126, 2009.

2 Ariful Azad, Grey Ballard, Aydin Buluç, James Demmel, Laura Grigori, Oded Schwartz, Sivan Toledo, and Samuel Williams. Exploiting Multiple Levels of Parallelism in Sparse Matrix-Matrix Multiplication. *SIAM J. Scientific Computing*, 38(6), 2016.

3 Grey Ballard, Aydin Buluç, James Demmel, Laura Grigori, Benjamin Lipshitz, Oded Schwartz, and Sivan Toledo. Communication optimal parallel multiplication of sparse random matrices. In *Proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures, (SPAA)*, pages 222–231, 2013.

4 Grey Ballard, Alex Druinsky, Nicholas Knight, and Oded Schwartz. Hypergraph Partitioning for Sparse Matrix-Matrix Multiplication. *TOPC*, 3(3):18:1–18:34, 2016.

5 Aydin Buluç and John R. Gilbert. The Combinatorial BLAS: design, implementation, and applications. *IJHPCA*, 25(4):496–509, 2011.

6 Aydin Buluç and John R. Gilbert. Parallel Sparse Matrix-Matrix Multiplication and Indexing: Implementation and Experiments. *SIAM J. Scientific Computing*, 34(4), 2012.

7 Keren Censor-Hillel, Petteri Kaski, Janne H. Korhonen, Christoph Lenzen, Ami Paz, and Jukka Suomela. Algebraic Methods in the Congested Clique. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 143–152, 2015.

8 Keren Censor-Hillel, Dean Leitersdorf, and Elia Turner. Sparse Matrix Multiplication in the Congested Clique model, 2018. `arXiv:arXiv:1802.04789`.

9 Don Coppersmith and Shmuel Winograd. Matrix Multiplication via Arithmetic Progressions. *J. Symb. Comput.*, 9(3):251–280, 1990.

10 Danny Dolev, Christoph Lenzen, and Shir Peled. "Tri, Tri Again": Finding Triangles and Small Subgraphs in a Distributed Setting - (Extended Abstract). In *Proceedings of the 26th International Symposium on Distributed Computing (DISC)*, pages 195–209, 2012.

11 Andrew Drucker, Fabian Kuhn, and Rotem Oshman. On the power of the congested clique model. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 367–376, 2014.

12 François Le Gall. Faster Algorithms for Rectangular Matrix Multiplication. In *Proceedings of the 53rd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 514–523, 2012.

13 François Le Gall. Powers of tensors and fast matrix multiplication. In *International Symposium on Symbolic and Algebraic Computation (ISSAC)*, pages 296–303, 2014.

14 François Le Gall. Further Algebraic Algorithms in the Congested Clique Model and Applications to Graph-Theoretic Problems. In *Proceedings of the 30th International Symposium on Distributed Computing (DISC)*, pages 57–70, 2016.

15 Francois Le Gall and Florent Urrutia. Improved Rectangular Matrix Multiplication using Powers of the Coppersmith-Winograd Tensor. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1029–1046, 2018.

16 Taisuke Izumi and François Le Gall. Triangle Finding and Listing in CONGEST Networks. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 381–389, 2017.

17 Haim Kaplan, Micha Sharir, and Elad Verbin. Colored intersection searching via sparse rectangular matrix multiplication. In *Proceedings of the 22nd ACM Symposium on Computational Geometry (SocG)*, pages 52–60, 2006.

18 Hartmut Klauck, Danupon Nanongkai, Gopal Pandurangan, and Peter Robinson. Distributed Computation of Large-scale Graph Problems. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 391–410, 2015.

19 Penporn Koanantakool, Ariful Azad, Aydin Buluç, Dmitriy Morozov, Sang-Yun Oh, Leonid Oliker, and Katherine A. Yelick. Communication-Avoiding Parallel Sparse-Dense Matrix-Matrix Multiplication. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 842–853, 2016.

20 Alfio Lazzaro, Joost VandeVondele, Jürg Hutter, and Ole Schütt. Increasing the Efficiency of Sparse Matrix-Matrix Multiplication with a 2.5D Algorithm and One-Sided MPI. *CoRR*, abs/1705.10218, 2017.

21 Christoph Lenzen. Optimal deterministic routing and sorting on the congested clique. In *The ACM Symposium on Principles of Distributed Computing (PODC)*, pages 42–50, 2013.

22 Danupon Nanongkai. Distributed approximation algorithms for weighted shortest paths. In *The 46th ACM Symposium on Theory of Computing (STOC)*, pages 565–573, 2014.

23 Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. On the Distributed Complexity of Large-Scale Graph Computations. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 405–414, 2018.

24 Edgar Solomonik, Maciej Besta, Flavio Vella, and Torsten Hoefler. Scaling betweenness centrality using communication-efficient sparse matrix multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 47:1–47:14, 2017.

25 Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.

26 Virginia Vassilevska Williams. Multiplying matrices faster than coppersmith-winograd. In *The 44th ACM Symposium on Theory of Computing (STOC)*, pages 887–898, 2012.

27 Raphael Yuster and Uri Zwick. Fast sparse matrix multiplication. *ACM Trans. Algorithms*, 1(1):2–13, 2005.

Large-Scale Distributed Algorithms for Facility Location with Outliers

Tanmay Inamdar
Department of Computer Science, The University of Iowa, Iowa, USA
tanmay-inamdar@uiowa.edu

Shreyas Pai
Department of Computer Science, The University of Iowa, Iowa, USA
shreyas-pai@uiowa.edu

Sriram V. Pemmaraju
Department of Computer Science, The University of Iowa, Iowa, USA
sriram-pemmaraju@uiowa.edu

Abstract

This paper presents fast, distributed, $O(1)$-approximation algorithms for metric facility location problems with outliers in the Congested Clique model, Massively Parallel Computation (MPC) model, and in the k-machine model. The paper considers *Robust Facility Location* and *Facility Location with Penalties*, two versions of the facility location problem with outliers proposed by Charikar et al. (SODA 2001). The paper also considers two alternatives for specifying the input: the input metric can be provided explicitly (as an $n \times n$ matrix distributed among the machines) or implicitly as the shortest path metric of a given edge-weighted graph. The results in the paper are:

- **Implicit metric**: For both problems, $O(1)$-approximation algorithms running in $O(\text{poly}(\log n))$ rounds in the Congested Clique and the MPC model and $O(1)$-approximation algorithms running in $\tilde{O}(n/k)$ rounds in the k-machine model.
- **Explicit metric**: For both problems, $O(1)$-approximation algorithms running in $O(\log \log \log n)$ rounds in the Congested Clique and the MPC model and $O(1)$-approximation algorithms running in $\tilde{O}(n/k)$ rounds in the k-machine model.

Our main contribution is to show the existence of Mettu-Plaxton-style $O(1)$-approximation algorithms for both Facility Location with outlier problems. As shown in our previous work (Berns et al., ICALP 2012, Bandyapadhyay et al., ICDCN 2018) Mettu-Plaxton style algorithms are more easily amenable to being implemented efficiently in distributed and large-scale models of computation.

2012 ACM Subject Classification Theory of computation → Facility location and clustering, Theory of computation → Distributed algorithms, Theory of computation → MapReduce algorithms, Theory of computation → Graph algorithms analysis

Keywords and phrases Distributed Algorithms, Clustering with Outliers, Metric Facility Location, Massively Parallel Computation, k-machine model, Congested Clique

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.5

Related Version [24], `http://arxiv.org/abs/1811.06494`

© Tanmay Inamdar, Shreyas Pai, and Sriram V. Pemmaraju;
licensed under Creative Commons License CC-BY
22nd International Conference on Principles of Distributed Systems (OPODIS 2018).
Editors: Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira; Article No. 5; pp. 5:1–5:16

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

1 Introduction

Metric Facility Location (in short, FacLoc) is a well-known combinatorial optimization problem used to model clustering problems. The input to the problem is a set F of *facilities*, an *opening cost* $f_i \geq 0$ for each facility $i \in F$, a set C of *clients*, and a metric space $(F \cup C, d)$ of *connection costs*, where $d(i, j)$ denotes the cost of client j connecting to facility i. The objective is to find a subset $F' \subseteq F$ of facilities to open so that the total cost of opening the facilities plus the cost of connecting all clients to open facilities is minimized. In other words, the quantity $cost(F') := \sum_{i \in F'} f_i + \sum_{j \in C} d(j, F')$ is minimized, where $d(j, F')$ denotes $\min_{i \in F'} d(i, j)$. FacLoc is NP-complete, but researchers have devised a number of approximation algorithms for the problem. For any $\alpha \geq 1$, an *α-approximation algorithm* for FacLoc finds in polynomial time, a subset $F' \subseteq F$ of facilities such that $cost(F') \leq \alpha \cdot cost(F^*)$, where F^* is an optimal solution to the given instance of FacLoc. There are several well-known $O(1)$-factor approximation algorithms for FacLoc including the primal-dual algorithm of Jain and Vazirani [25] and the greedy algorithm of Mettu and Plaxton [33]. The best approximation factor currently achieved by an algorithm for FacLoc is 1.488 [30]. More recently, motivated by the need to solve FacLoc and other clustering problems on extremely large inputs, researchers have proposed distributed and parallel approximation algorithms for these problems. See for example [14, 15] for clustering algorithms in systems such as MapReduce [11] and Pregel [32] and [4] for clustering algorithms in the *k-machine model*. Clustering algorithms [38] have also been designed for streaming models of computation [1].

Outliers can pose a problem for many statistical methods. For clustering problems, a few outliers can have an outsized influence on the optimal solution, forcing the opening of costly extra facilities or leading to poorer service to many clients. Versions of FacLoc that are robust to outliers have been proposed by Charikar et al. [10], where the authors also present $O(1)$-approximation algorithms for these problems. Specifically, Charikar et al. [10] propose two versions of FacLoc that are robust to outliers:

Robust FacLoc: In addition to F, C, opening costs $\{f_i | i \in F\}$, and metric d, we are also given an integer $0 \leq p \leq |C|$, that denotes the *coverage requirement*. The objective is to find a solution (C', F'), where $F' \subseteq F$, $C' \subseteq C$, with $|C'| \geq p$, and

$$\mathsf{cost}(C', F') := \sum_{i \in F'} f_i + \sum_{j \in C'} d(j, F')$$

is minimized over all (F', C'), where $|C'| \geq p$.

FacLoc with Penalties: In addition to F, C, opening costs $\{f_i | i \in F\}$, and metric d, we are also given penalties $p_j \geq 0$ for each client $j \in C$. The objective is to find a solution (C', F'), where $F' \subseteq F$, and $C' \subseteq C$, such that,

$$\mathsf{cost}(C', F') := \sum_{i \in F'} f_i + \sum_{j \in C'} d(j, F') + \sum_{j \in C \setminus C'} p_j$$

is minimized over all (C', F').

In this paper we present distributed $O(1)$-approximation algorithms for Robust FacLoc and FacLoc with Penalties in several models of large-scale distributed computation. As far as we know, these are the first distributed algorithms for versions of FacLoc that are robust to outliers. In distributed settings, the complexity of the problem can be quite sensitive to the manner in which input is specified. We consider two alternate ways of specifying the input to the problem.

Explicit metric: The metric d is specified *explicitly* as a $|F| \times |C|$ matrix distributed among the machines of the underlying communication network. This explicit description of the metric assumes that the $|F| \times |C|$ matrix fits in the total memory of all machines combined.

Implicit metric: In this version, the metric is specified *implicitly* – as the shortest path metric of a given edge-weighted graph whose vertex set is $C \cup F$; we call this the *metric graph*. The reason for considering this alternate specification of the metric is that it can be quite compact; the graph specifying the metric can be quite sparse (e.g., having $O(|F| + |C|)$ edges). Thus, in settings where $|F| \cdot |C|$ is excessively large, but $|F| + |C|$ is not, this is a viable option.

For the facility location problems considered in this paper, when the input metric is explicitly specified, the biggest challenge is solving the *maximal independent set (MIS)* problem efficiently. When the input metric is implicitly specified, the biggest challenge is to efficiently learn just enough of the metric space. Thus, changing the input specification changes the main challenge in a fundamental way and consequently we obtain very different results for the two alternate input specifications.

Our algorithms run in 3 models of distributed computation, which we now describe specifically in the context of facility location problems. All three models are synchronous message passing model.

Congested Clique model: The Congested Clique model was introduced by Lotker et al. [31] and then extensively studied in recent years [17, 18, 16, 9, 23, 26, 34, 13, 12, 37, 29]. In this model, the underlying communication network is a clique and the number of nodes in this clique equals $|F|+|C|$. In each round, each node performs local computation based on the information it holds and then sends a (possibly distinct) $O(\log n)$-size message to each of the remaining nodes. Initially, each node hosts a facility or a client and the node hosting facility i knows the opening cost f_i and the node hosting client j knows the penalty p_j for the FacLoc with penalties problem. In the explicit metric setting, the node hosting facility i knows all the connection costs $d(i, j)$ to all clients $j \in C$. Similarly, the node hosting client j knows all connection costs $d(i, j)$ to all facilities $i \in F$. In the implicit metric setting, the node hosting a facility or client knows the edges of the metric graph incident on that facility or client. We call this input distribution *vertex-centric* because each node is responsible for the local input of a facility or client. The vertex-centric assumption can be made without loss of generality because an adversarially (but evenly) distributed input can be redistributed in a vertex-centric manner among the nodes in constant rounds using Lenzen's routing protocol [29].

Massively Parallel Computation (MPC) model: The MPC model was introduced in [27] and variants of this model were considered in [19, 5, 2]. It can be viewed as a clean abstraction of the MapReduce model. We are given k machines, each with S words of space and the input is distributed in a vertex-centric fashion among the machines, the only difference being that machines can host multiple facilities and clients (provided they fit in memory). Let I be the total input size. Typically, we require k and S to each be sublinear in I, that is $O(I^{1-\varepsilon})$ for some $\varepsilon > 0$. We also require that the total memory not be too much larger than needed for the input, i.e., $k \times S = O(I)$. In each round, each machine sends and receives a total of $O(S)$ words of information because it is the volume of information that will fit into its memory. In our work we consider

MPC algorithms with memory $S = \tilde{O}(n)$ [1] where $n = |F| = |C|$. In the explicit metric setting, since $I = O(n^2)$, even if we assume $S = \tilde{O}(n)$, k and S are still strictly sublinear in I. But in the implicit metric setting, if we assume $S = \tilde{O}(n)$ then the memory may not be strictly sublinear in the input size when the input graph is sparse, having $O(n)$ edges for example. Therefore, our algorithms are not strictly MPC algorithms when the input is sparse. Similar to the Congested Clique model, we can assume that the input is distributed in a vertex-centric manner without loss of generality, due to the nature of communication in each round and the fact that $S = \Omega(n)$.

k-machine model: The k-machine model was introduced in [28] and further studied in [36]. This model abstracts essential features of systems such as Pregel [32] and Giraph (see `http://giraph.apache.org/`) that have been designed for large-scale graph processing. We are given k machines and the input is distributed among the machines. In [28], the k-machine model is used to solve graph problems and they assume a *random vertex partition* distribution of the input graph among the k machines. In other words, each vertex along with its incident edges is provided to one of the k machines chosen uniformly at random. The corresponding assumption for facility location problems would be that each facility and each client is assigned uniformly at random to one of the k machines. Facility $i \in F$ comes with its opening cost f_i and client $j \in C$ comes with its penalty p_j for the FacLoc with penalties problem. In the explicit metric setting, each facility $i \in F$ comes with connections costs $d(i, j)$ for all $j \in C$ whereas in the implicit metric setting facility i comes along with the edges of the metric graph incident on it. Similarly for each client $j \in C$. In each round, each machine can send a (possibly distinct) size-B message to each of the remaining $k - 1$ machines. Typically, B is assumed to be $\text{poly}(\log n)$ bits [28].

The Congested Clique model does not directly model settings of large-scale computation because in this model the number of nodes in the underlying communication network equals the number of vertices in the input graph. However, fast Congested Clique algorithms can usually be translated (sometimes automatically) to fast MPC and k-machine algorithms. So the Congested Clique algorithms in this paper are important stepping stones towards more complex MPC and k-machine algorithms [20, 28]. The MPC model and the k-machine model are quite similar. Even though the k-machine model is specified with a per-edge bandwidth constraint of B bits, it can be equivalently described with a per-machine bandwidth constraint of $k \cdot B$ bits that can be sent and received in each round. Thus setting $k \cdot B = S$ makes the k-machine model and MPC model equivalent in their bandwidth constraint. Despite their similarities, it is useful to think about both models due to differences in how they are parameterized and how these parameters affect the running times of algorithms in these models. For example, in the MPC model, usually one starts by picking S as a sublinear function of the input size n. This leads to the number of machines being fixed and the running time of the algorithm is expressed as a function of n. In the k-machine model B is usually fixed at $\text{poly}(\log n)$ and the running time of the algorithm is expressed as a function of n and k. This helps us understand how the running time changes as we increase k. For example, algorithms with running times of the form $O(n/k)$ exhibit a linear speedup as k increases, whereas algorithms with running time of the form $O(n/k^2)$ indicating a quadratic speedup [35].

[1] Throughout the paper, we use $\tilde{O}(f(n))$ as a shorthand for $O(f(n) \cdot \text{poly}(\log n))$ and $\tilde{\Omega}(f(n))$ as a shorthand for $\Omega(f(n)/\text{poly}(\log n))$.

1.1 Main Results

In order to obtain $O(1)$-approximation algorithms for Robust FacLoc and FacLoc with Penalties, Charikar et al. [10] propose modifications to the primal-dual approximation algorithm for FacLoc due to Jain and Vazirani [25]. The problem with using this approach for our purposes is that it seems difficult obtain fast *distributed* algorithms using the Jain-Vazirani approach. For example, obtaining a *sublogarithmic* round $O(1)$-approximation for FacLoc in the Congested Clique model using this approach seems difficult. However, as established in our previous work [21, 4, 8] and in [15] the greedy algorithm of Mettu and Plaxton [33] for FacLoc seems naturally suited for fast distributed implementation.

The first contribution of this paper is to show that $O(1)$-approximation algorithms to Robust FacLoc and FacLoc with Penalties can *also* be obtained by using variants of the Mettu-Plaxton greedy algorithm. Our second contribution is to show that by combining ideas from earlier work [21, 4] with some new ideas, we can efficiently implement distributed versions of the variants of the Mettu-Plaxton algorithm for Robust FacLoc and FacLoc with Penalties. The specific results we obtain for the two versions of input specification are as follows. For simplicity of exposition, we assume $|C| = |F| = n$.

- **Implicit metric**: For both problems, we present $O(1)$-approximation algorithms running in $O(\text{poly}(\log n))$ rounds in the Congested Clique and the MPC model. Assuming the metric graph has m edges, the input size is $\Theta(m + n)$ and we use $\tilde{O}(m/n)$ machines each with memory $\tilde{O}(n)$. In the k-machine model, we present $O(1)$-approximation algorithms running in $\tilde{O}(n/k)$ rounds.
- **Explicit metric**: For both problems, we present extremely fast $O(1)$-approximation algorithms, running in $O(\log \log \log n)$ rounds, in the Congested Clique and the MPC model. The input size is $\Theta(n^2)$ and we use n machines each with memory $\tilde{O}(n)$ in the MPC model. In the k-machine model, we present $O(1)$-approximation algorithms running in $\tilde{O}(n/k)$ rounds.

Due to space constraints we only describe our distributed implementations for the Robust FacLoc algorithm and omit most of the technical proofs. The full version with all the technical details appears in [24].

2 Sequential Algorithms for Facility Location with Outliers

We first describe the greedy sequential algorithm of Mettu and Plaxton [33] (Algorithm 1) for the Metric FacLoc problem which will serve as a building block for our algorithms for Robust FacLoc and the FacLoc with Penalties discussed in this section. The algorithm first computes a "radius" r_i for each facility $i \in F$ and it then greedily picks facilities to open in non-decreasing order of radii provided no previously opened facility is too close. The "radius" of a facility i is the amount that each client is charged for the opening of facility i. Clients pay towards this charge after paying towards the cost of connecting to facility i; clients that have a large connection cost to i pay nothing towards this charge. It is shown in [33] using a charging argument that Algorithm 1 is 3-approximation for the Metric FacLoc problem. Later on, [3] gave a primal-dual analysis, showing the same approximation guarantee, by comparing the cost of the solution to a dual feasible solution. We use the latter analysis approach as it can be easily modified to work for the algorithms with outliers.

Algorithm 1: FacilityLocationMP(F, C).

```
/* Radius Computation Phase: */
1 For each i ∈ F, compute r_i ≥ 0, satisfying f_i = ∑_{j∈C} max{0, r_i − c_ij}.
/* Greedy Phase: */
2 Sort and renumber facilities in the non-decreasing order of r_i.
3 F' ← ∅                                          ▷ Solution set
4 for i = 1, 2, ... do
5   | if there is no facility in F' within distance 2r_i from i then
6   |   | F' ← F' ∪ {i}
7   | end
8 end
9 Connect each client j to its closest facility in F'.
```

2.1 Robust Facility Location

Since we use the primal dual analysis of [3] to get a bounded approximation factor, we need to address the fact that the standard linear programming relaxation for Robust FacLoc has unbounded integrality gap. To fix this we modify the instance in a similar manner to [10]. Let (C^*, F^*) be a fixed optimal solution, and let $i_* \in F$ be a facility in that solution with the maximum opening cost f_{i_*}. We begin by assuming that we are given a facility, say i_e with opening cost f_{i_e}, such that, $f_{i_*} \leq f_{i_e} \leq \alpha f_{i_*}$, where $\alpha \geq 1$ is a constant. Now, we modify the original instance by changing the opening costs of the facilities as follows.

$$f'_i = \begin{cases} +\infty & \text{if } f_i > f_{i_e} \\ 0 & \text{if } i = i_e \\ f_i & \text{otherwise} \end{cases}$$

Note that we can remove the facilities with opening cost $+\infty$ without affecting the cost of an optimal solution, and hence we assume that w.l.o.g. all the modified opening costs f'_i are finite.

Let (C^*_e, F^*_e) be an optimal solution for this modified instance, and let $\mathsf{cost}_e(C^*_e, F^*_e)$ be its cost using the modified opening costs. Observe that without loss of generality, we can assume that $i_e \in F^*_e$, since its opening cost f'_{i_e} equals 0. We obtain the following lemma and its simple corollary.

▶ **Lemma 1.** $\mathsf{cost}_e(C^*_e, F^*_e) \leq \mathsf{cost}(C^*, F^*)$.

▶ **Corollary 2.** *Let (C'_e, F'_e) be a feasible solution for the instance with modified facility opening costs, such that, $\mathsf{cost}_e(C'_e, F'_e) \leq \beta \cdot \mathsf{cost}_e(C^*_e, F^*_e) + \gamma \cdot f_{i_e}$ (where $\beta \geq 1, \gamma \geq 0$). Then, (C'_e, F'_e) is a $\beta + \alpha \cdot (\gamma + 1)$ approximation for the original instance.*

To efficiently find a facility i_e satisfying $f_{i_*} \leq f_{i_e} \leq \alpha f_{i_*}$, we partition the facilities into sets where each set contains facilities with opening costs from the range $\big[(1+\varepsilon)^i, (1+\varepsilon)^{i+1}\big)$. Iterating over all such ranges, and choosing a facility with highest opening cost from that range, we are guaranteed to find a facility i_e such that, $f_{i_*} \leq f_{i_e} \leq (1+\varepsilon) f_{i_*}$ [2]. The total

[2] An alternative approach would be to consider each facility one-by-one as a candidate, but for an efficient distributed implementation we can only afford $O(\log n)$ distinct guesses.

number of such iterations will be $O(\log_{1+\varepsilon} \frac{f_{\max}}{f_{\min}})$, where $f_{\max}$ is the largest opening cost, and $f_{\min}$ is the smallest non-zero opening cost. Assuming that every individual item in the input (e.g., facility opening costs, connection costs, etc.) can each be represented in $O(\log n)$ bits and that ε is a constant, this amounts to $O(\log n)$ iterations.

Our facility location algorithm is described in Algorithm 2. This algorithm can be thought of as running $O(\log n)$ separate instances of a modified version of the original Mettu-Plaxton algorithm (Algorithm 1), where in each instance of the Mettu-Plaxton algorithm, the algorithm is terminated as soon as the number of outlier clients drops below the required number, following which there is some post-processing.

We abuse the notation slightly, and denote by (C', F') the solution returned by the algorithm, i.e., the solution (C'_t, F'_t) corresponding to the iteration t of the outer loop that results in a minimum cost solution. Similarly, we denote by i_e a facility chosen in line 2 in the iteration corresponding to this iteration t.

▶ **Theorem 3.** $\mathsf{cost}_e(C', F') \leq 3 \cdot \mathsf{cost}_e(C^*_e, F^*_e) + f_{i_e}$

Applying Corollary 2 with $\alpha = 1 + \varepsilon, \beta = 3, \gamma = 1$ yields the following approximation guarantee.

▶ **Theorem 4.** *The solution returned by Algorithm 2 is a $5 + \varepsilon$ approximation to the Robust* FacLoc *problem.*

2.2 Facility Location with Penalties

For the penalty version, each client j comes with a penalty p_j which is the cost we pay if we make j an outlier. Therefore, the radius computation for a facility changes because if a facility i is asking client j to contribute more than $p_j - c_{ij}$ then it is cheaper for j to mark itself as an outlier and pay its penalty. Therefore, for each facility $i \in F$, let $r_i \geq 0$ be a value such that $f_i = \sum_{j \in C} \max\{\min\{r_i - c_{ij}, p_j - c_{ij}\}, 0\}$, if it exists. Notice that if for a facility $i \in F$, such an r_i does not exist, then it must be the case that for all $j \in C$, $p_j \leq c_{ij}$. That is, it is for any client, it is cheaper to pay the penalty than to connect it to this facility. Therefore, removing such a facility from consideration does not affect the cost of any solution, and hence we assume that for all $i \in F$, an $r_i \geq 0$ exists such that $f_i = \sum_{j \in C} \max\{\min\{r_i - c_{ij}, p_j - c_{ij}\}, 0\}$. The algorithm for FacLoc with Penalties is shown in Algorithm 3.

A primal-dual analysis of Algorithm 3 leads to the following upper bound.

▶ **Theorem 5.** $\mathsf{cost}(C', F') \leq 3 \cdot \mathsf{cost}(C^*, F^*)$.

3 Distributed Robust Facility Location: Implicit Metric

We first present our k-machine algorithm for Distributed Robust FacLoc in the implicit metric setting and derive the Congested Clique as a special case for $k = n$. We then describe how to implement the algorithm in the MPC model.

3.1 The k-Machine Algorithm

In this section we show how to implement the sequential algorithms for the Robust FacLoc in the k-machine model. To do this we first need to establish some primitives and techniques. These have largely appeared in [4]. Then we will provide details for implementing the Robust FacLoc algorithm in the k-machine model.

Algorithm 2: RobustFacLoc(F, C, p).

/* Recall: $\ell \coloneqq |C| - p$ */
1 **for** $t = 0, \ldots, O(\log n)$ **do**
2 Let $i_e \in F$ be the most expensive facility from the facilities with opening costs in the range $[(1+\varepsilon)^t, (1+\varepsilon)^{t+1})$ for some small constant $\varepsilon > 0$
3 Modify the facility opening costs to be

$$f'_i = \begin{cases} +\infty & \text{if } f_i > f_{i_e} \\ 0 & \text{if } i = i_e \\ f_i & \text{otherwise} \end{cases}$$

/* Radius Computation Phase: */
4 For each $i \in F$, compute $r_i \geq 0$, satisfying $f'_i = \sum_{j \in C} \max\{0, r_i - c_{ij}\}$.
/* Greedy Phase: */
5 Sort and renumber facilities in the non-decreasing order of r_i.
6 Let $C' \leftarrow \emptyset$, $F' \leftarrow \emptyset$, $O' \leftarrow C$
7 Let $F_0 \leftarrow \emptyset$
8 **for** $i = 1, 2, \ldots$ **do**
9 **if** *there is no facility in* F' *within distance* $2r_i$ *from* i **then**
10 $F' \leftarrow F' \cup \{i\}$
11 **end**
12 $F_i \leftarrow F_{i-1} \cup \{i\}$
13 Let C_i denote the set of clients that are within distance r_i
14 $C' \leftarrow C' \cup C_i$, $O' \leftarrow O' \setminus C_i$.
15 **if** $|O'| \leq \ell$ **then break**
16 **end**
/* Outlier Determination Phase: */
17 **if** $|O'| > \ell$ **then**
18 Let $O_1 \subseteq O'$ be a set of $|O'| - \ell$ clients that are closest to facilities in F'.
19 $C' \leftarrow C' \cup O_1$, $O' \leftarrow O' \setminus O_1$.
20 **end**
21 **else if** $|O'| < \ell$ **then**
22 Let $O_2 \subseteq C'$ be the set of $\ell - |O'|$ clients with largest distance to open facilities F'.
23 $C' \leftarrow C' \setminus O_2$, $O' \leftarrow O' \cup O_2$.
24 **end**
25 Let $(C'_t, F'_t) \leftarrow (C', F')$
26 **end**
27 **return** Return (C'_t, F'_t) with the minimum cost.

Since the input metric is only implicitly provided, as an edge-weighted graph, a key primitive that we require is computing shortest path distances to learn parts of the metric space. To this end, the following lemma shows that we can solve the Single Source Shortest Paths (SSSP) problem efficiently in the k-machine model.

▶ **Lemma 6** (Corollary 1 in [4]). *For any $0 < \varepsilon \leq 1$, there is a deterministic $(1+\varepsilon)$-approximation algorithm in the k-machine model for solving the SSSP problem in undirected graphs with non-negative edge-weights in $O((n/k) \cdot poly(\log n)/poly(\varepsilon))$ rounds.*

Algorithm 3: PenaltyFacLoc(F, C, p).

```
   /* Radius Computation Phase: */
1  Compute r_i for each i ∈ F satisfying f_i = Σ_{j∈C} max{min{r_i − c_ij, p_j − c_ij}, 0}.
   /* Greedy Phase: */
2  Sort and renumber facilities in the non-decreasing order of r_i.
3  C' ← ∅,   F' ← ∅,   O' ← ∅.
4  for i = 1, 2, ... do
5  |  if there is no facility in F' within distance 2r_i from i then
6  |  |  F' ← F' ∪ {i}
7  |  end
8  end
   /* Outlier Determination Phase: */
9  for each client j do
10 |  Let i be the closest facility to j in F'
11 |  if c_ij ≤ p_j then C' ← C' ∪ {j}
12 |  else O' ← O' ∪ {j}
13 end
14 return (C', F') as the solution.
```

Algorithm 4: RadiusComputation Algorithm.

1. **Neighborhood-Size Computation.** Each machine m_j computes $q_i(v)$, for all integers $i \geq 0$ and for all vertices $v \in H(m_j)$.
2. **Local Computation.** Each machine m_j computes $\tilde{r}_v$ locally, for all vertices $v \in H(m_j)$. (Recall that $\tilde{r}_v := (1+\varepsilon)^{t-1}$ where $t \geq 1$ is the smallest integer for which $\sum_{i=0}^{t} q_i(v) \cdot ((1+\varepsilon)^{i+1} - (1+\varepsilon)^i) > f_v$.)

In addition to SSSP, our algorithms require an efficient solution to a more general problem that we call *Multi-Source Shortest Paths* (in short, MSSP). The input is an edge-weighted graph $G = (V, E)$, with non-negative edge-weights, and a set $T \subseteq V$ of sources.

For MSSP, the output is required to be, for each vertex v, the distance $d(v, T)$ (i.e., $\min\{d(v, u) \mid u \in T\}$) and the vertex $v^* \in T$ that realizes this distance. The following lemma shows that we can solve this problem efficiently in the k-machine model.

▶ **Lemma 7** (Lemma 4 in [4]). *Given a set $T \subseteq V$ of sources known to the machines (i.e., each machine m_j knows $T \cap H(m_j)$), we can, for any value $0 \leq \varepsilon \leq 1$, compute a $(1+\varepsilon)$-approximation to MSSP in $\tilde{O}(1/poly(\varepsilon) \cdot n/k)$ rounds, w.h.p. Specifically, after the algorithm has ended, for each $v \in V \setminus T$, the machine m_j that hosts v knows a pair $(u, \tilde{d}) \in T \times \mathbb{R}^+$, such that $d(v, u) \leq \tilde{d} \leq (1+\varepsilon) \cdot d(v, T)$.*

Using the primitives described above, [4] show that it is possible to compute approximate radius values efficiently in the k-machine model. The algorithm is described here, see the full version [24] for details on the implementation of this algorithm.

For any facility or client v and for any integer $i \geq 1$, let $q_i(v)$ denote $|B(v, (1+\varepsilon)^i)|$, the size of the neighborhood of v within distance $(1+\varepsilon)^i$.

Therefore, we get the following lemma the proof of which can be found in Section 4 of [4].

▶ **Lemma 8.** *For each facility $v \in F$ it is possible to compute an approximate radius $\tilde{r}_v$ in $\tilde{O}(n/k)$ rounds of the k-machine model such that $\frac{r_v}{(1+\varepsilon)^2} \leq \tilde{r}_v \leq (1+\varepsilon)^2 r_v$ where r_v is the actual radius of v satisfying $f_v = \sum_{u \in B(v,r_v)}(r_v - d(v,u))$.*

The greedy phase is implemented by discretizing the radius values computed in the first phase which results in $O(\log_{1+\varepsilon} n)$ distinct categories. Note that in each category, the order in which we process the facilities does not matter as it will only add an extra $(1+\varepsilon)$ factor to the approximation ratio. This reduces the greedy phase to computing a *maximal independent set (MIS)* on a suitable intersection graph for each category i where the vertices are the facilities in the ith category and there is an edge between two vertices if they are within distance $2(1+\varepsilon)^i$ of each other.

Finding such an MIS requires $O(\log n)$ calls to a subroutine that solves MSSP [39] and since our implementation of MSSP only returns approximate distances, what we really compute is a relaxed version of an MIS called an (ε, d)-MIS in [4].

▶ **Definition 9** ((ε, d)-approximate MIS)**.** For an edge-weighted graph $G = (V, E)$, and parameters $d, \varepsilon > 0$, an (ε, d)-approximate MIS is a subset $I \subseteq V$ such that

1. For all distinct vertices $u, v \in I$, $d(u, v) \geq \frac{d}{1+\varepsilon}$.
2. For any $u \in V \setminus I$, there exists a $v \in I$ such that $d(u, v) \leq d \cdot (1+\varepsilon)$.

The work in [4] gives an algorithm that efficiently computes an approximate MIS of an induced subgraph $G[W]$ of G for any vertex set W in the k-machine model.

▶ **Lemma 10.** *We can find an $(O(\varepsilon), d)$-approximate MIS I of $G[W]$ whp in $\tilde{O}(n/k)$ rounds.*

We are now ready to describe the k-machine model implementation of Algorithm 2.

Our k-machine model implementation of the Robust FacLoc algorithm is summarized in Algorithm 5. The correctness proof is similar to that of Algorithm 2 but is complicated by the fact that we compute $(1+\varepsilon)$-approximate distances instead of exact distances. Again, as in the analysis of the sequential algorithm, we abuse the notation so that (i) (C', F') refers to a minimum-cost solution returned by the algorithm, (ii) i_e refers to the facility chosen in the line 2 of the algorithm, and (iii) the modified instance with original facility costs. This analysis appears in the full version [24], and as a result we get the following theorem.

▶ **Theorem 11.** *In $\tilde{O}(poly(1/\varepsilon) \cdot n/k)$ rounds, whp, Algorithm 5 finds a factor $5 + O(\varepsilon)$ approximate solution (C', F') to the Robust FacLoc problem for any constant $\varepsilon > 0$.*

3.2 The Congested Clique and MPC Algorithms

The algorithm for Congested Clique is essentially the same as the k-machine model algorithm with $k = n$. The only technical difference is that in the k-machine model, the input graph vertices are randomly partitioned across the machines. This means that even though there are n vertices and n machines, a single machine may be hosting multiple vertices. It is easy to see that the Congested Clique model, in which each machine holds exactly one vertex can simulate the k-machine algorithm with no overhead in rounds. Therefore, by substituting $k = n$ in the running time of Theorem 11, we get the following result.

▶ **Theorem 12.** *In $O(\text{poly} \log n)$ rounds of Congested Clique, whp, we can find a factor $5 + O(\varepsilon)$ approximate solution to the Robust FacLoc problem for any constant $\varepsilon > 0$.*

Now we focus on the implementing the MPC algorithm. The first crucial observation is that Algorithm 5 reduces the task of finding an approximate solution to the Robust FacLoc

Algorithm 5: RobustFacLocDist(F, C, p).

```
/* Recall ℓ := |C| − p                                                    */
1  for t = 1, ..., O(log n) do
2      Let i_e ∈ F be a most expensive facility from the facilities with opening costs in
         the range [(1 + ε)^t, (1 + ε)^{t+1})
3      Modify the facility opening costs to be
```

$$f'_i = \begin{cases} +\infty & \text{if } f_i > f_{i_e} \\ 0 & \text{if } i = i_e \\ f_i & \text{otherwise} \end{cases}$$

```
       /* Radius Computation Phase:                                          */
4      Call the RadiusComputation algorithm (Algorithm 4) to compute approximate
         radii.
       /* Greedy Phase:                                                      */
5      Let F' = ∅, C' = ∅, O' = C
6      for i = 0, 1, 2, ... do
7          Let W be the set of vertices w ∈ F across all machines with r̃_w = r̃ = (1 + ε)^i
8          Using Lemma 7, remove all vertices from W within approximate distance
             2(1 + ε)^3 · r̃ from F'
9          I ← ApproximateMIS(G, W, 2(1 + ε)^3 · r̃, ε)
10         F' ← F' ∪ I
11         Using Lemma 7, move from O' to C' all vertices that are within distance
             (1 + ε) · r̃ from F_i, the set of facilities processed up to iteration i
12         if |O'| ≤ ℓ then break
13     end
       /* Outlier Determination Phase:                                       */
14     if |O'| > ℓ then
15         Using Lemma 7 find O_1 ⊆ O', a set of |O'| − ℓ clients that are closest to
             facilities in F'.
16         C' ← C' ∪ O_1,    O' ← O' \ O_1.
17     end
18     else if |O'| < ℓ then
19         Using Lemma 7 find O_2 ⊆ C \ O', a set of (ℓ − |O'|) clients that are farthest
             away from facilities in F'
20         C' ← C' \ O_2,    O' ← O' ∪ O_2.
21     end
22     Let (C'_t, F'_t) ← (C', F')
23 end
24 return (C'_t, F'_t) with a minimum cost
```

problem in the implicit metric setting to poly $\log n$ calls to a $(1+\varepsilon)$-approximate SSSP subroutine along with some local bookkeeping. Therefore, all we need to do is efficiently implement an approximate SSSP algorithm in the MPC model.

The second fact that helps us is that Becker et al. [6] provide a distributed implementation of their approximate SSSP algorithm in the Broadcast Congested Clique (BCC) model. The

BCC model is the same as the Congested Clique model but with the added restriction that nodes can only broadcast messages in each round. Therefore we get the following simulation theorem, which follows almost immediately from Theorem 3.1 of [7].

▶ **Theorem 13.** *Let $\mathcal{A}$ be a T round BCC algorithm that uses $\tilde{O}(n)$ local memory at each node. One can simulate $\mathcal{A}$ in the MPC model in $O(T)$ rounds using $\tilde{O}(n)$ memory per machine.*

In any T round BCC algorithm, each vertex will receive $O(n \cdot T)$ distinct messages. The approximate SSSP algorithm of Becker et al. [6] runs in $O(\text{poly} \log n / \text{poly}(\varepsilon))$ rounds and therefore, uses $\tilde{O}(n)$ memory per node to store all the received messages (and for local computation). Therefore, we get the following theorem.

▶ **Theorem 14.** *In $O(\text{poly} \log n)$ rounds of MPC, whp, we can find a factor $5 + O(\varepsilon)$ approximate solution to the Robust* FacLoc *problem for any constant $\varepsilon > 0$.*

4 Distributed Robust Facility Location: Explicit Metric

For the k-machine model implementation, the implicit metric algorithm from the previous section also provides a similar guarantee for the explicit metric setting and hence we do not discuss it separately in this section.

4.1 The Congested Clique Algorithm

The work in [21] presents a Congested Clique algorithm that runs in expected $O(\log \log n)$ rounds and computes an $O(1)$-approximation to FacLoc. This is improved exponentially in [22] which presents an $O(1)$-approximation algorithm to FacLoc running in $O(\log \log \log n)$ rounds whp. The algorithms in [21] and in [22] are essentially the same with one key difference. They both reduce the problem of solving FacLoc in the Congested Clique model to the ruling set problem. Specifically, showing that if a t-ruling set can be computed in T rounds, then an $O(t)$-approximation to FacLoc can be computed in $O(T)$ rounds. In [21] a 2-ruling set is computed in expected $O(\log \log n)$ rounds, whereas in [22] it is computed in $O(\log \log \log n)$ rounds whp.

The algorithm for computing an $O(1)$-approximation to Robust FacLoc (see Section 2.1) is essentially the FacLoc algorithm in [21, 22], but with an outer loop that runs $O(\log n)$ times. In each iteration of this outer loop, we modify the facility opening costs in a certain way and solve FacLoc on the resulting instance. Thus we have $O(\log n)$ instances of FacLoc to solve and via the reduction in [21, 22], we have $O(\log n)$ independent instances of the ruling set problem to solve. Here we show that $O(\log n)$ independent instances of the $O(\log \log \log n)$-round 2-ruling set algorithm in [22] can be executed in parallel in the Congested Clique model, still in $O(\log \log \log n)$ rounds whp. To be precise, suppose that the input consists of $c = O(\log n)$ graphs $G_1 = (V, E_1), G_2 = (V, E_2), \ldots, G_c = (V, E_c)$.

▶ **Theorem 15.** *2-ruling sets for all graphs G_i, $1 \leq i \leq c$, can be computed in $O(\log \log \log n)$ rounds whp.*

The theorem above and the discussion preceding it leads to the following theorem.

▶ **Theorem 16.** *There is an $O(1)$-approximation algorithm in the Congested Clique model for Robust* FacLoc*, running in $O(\log \log \log n)$ rounds whp.*

4.2 The MPC Algorithm

We now utilize the Congested Clique algorithm for Robust FacLoc to design an MPC model algorithm for Robust FacLoc, also running in $O(\log\log\log n)$ rounds whp. Since each vertex has explicit knowledge of n distances, the overall memory is $O(n^2)$ words. Since the memory of each machine is $\tilde{O}(n)$, the number of machines will be $\tilde{O}(n)$ as well. Therefore, we can simulate the algorithm from the preceding section using Theorem 3.1 of [7] in the MPC model. We summarize our result in the following theorem.

▶ **Theorem 17.** *There is an $O(1)$-approximation algorithm for Robust* FacLoc *that can be implemented in the MPC model with $\tilde{O}(n)$ words per machine in $O(\log\log\log n)$ rounds whp.*

5 Conclusion and Open Questions

This paper presents fast $O(1)$-factor distributed algorithms for Facility Location problems that are robust to outliers. These algorithms run in the Congested Clique model and two models of large-scale computation, namely, the MPC model and the k-machine model. As far as we know these are the the first such algorithms for these important clustering problems.

Fundamental questions regarding the optimality of our results remain open. In the explicit metric setting, we present algorithms in the Congested Clique model and the MPC model that run in $O(\log\log\log n)$ rounds. While these may seem extremely fast, it is not clear that they are optimal. Via the results of Drucker et al. [13], it seems like showing a non-trivial lower bound in the Congested Clique model is out of the question for now. So a tangible question one can ask is whether we can further improve the running time of the 2-ruling set algorithm in the Congested Clique model, possibly solving it in $O(\log^* n)$ or even $O(1)$ rounds. This would immediately imply a corresponding improvement in the running time of our Congested Clique and MPC model algorithms in the explicit metric setting.

All the k-machine algorithms we present in the paper run in $\tilde{O}(n/k)$ rounds. It is unclear if this is optimal. In previous work [4], we showed a lower bound of $\tilde{\Omega}(n/k)$ in the implicit metric setting, assuming that in the output to facility location problems every open facility needed to know all clients that connect to it. The lower bound heavily relies on the implicit metric and the output requirement assumptions. However, even if we relax both of these assumptions, i.e., we work in the explicit metric setting and only ask that every client know the facility that will serve it, we still seem to be unable to get over the $\tilde{O}(n/k)$ barrier.

References

1 Noga Alon, Yossi Matias, and Mario Szegedy. The Space Complexity of Approximating the Frequency Moments. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 20–29, New York, NY, USA, 1996. ACM. doi:10.1145/237814.237823.

2 Alexandr Andoni, Aleksandar Nikolov, Krzysztof Onak, and Grigory Yaroslavtsev. Parallel Algorithms for Geometric Graph Problems. In *Proceedings of the Forty-sixth Annual ACM Symposium on Theory of Computing*, STOC '14, pages 574–583, New York, NY, USA, 2014. ACM. doi:10.1145/2591796.2591805.

3 Aaron Archer, Ranjithkumar Rajagopalan, and David B. Shmoys. Lagrangian Relaxation for the k-Median Problem: New Insights and Continuity Properties. In Giuseppe Di Battista and Uri Zwick, editors, *Algorithms - ESA 2003: 11th Annual European Symposium, Budapest, Hungary, September 16-19, 2003. Proceedings*, pages 31–42, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. doi:10.1007/978-3-540-39658-1_6.

4 Sayan Bandyapadhyay, Tanmay Inamdar, Shreyas Pai, and Sriram V. Pemmaraju. Near-Optimal Clustering in the k-machine model. In *Proceedings of the 19th International Conference on Distributed Computing and Networking, ICDCN 2018, Varanasi, India, January 4-7, 2018*, pages 15:1–15:10, 2018. doi:10.1145/3154273.3154317.

5 Paul Beame, Paraschos Koutris, and Dan Suciu. Communication Steps for Parallel Query Processing. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '13, pages 273–284, New York, NY, USA, 2013. ACM. doi:10.1145/2463664.2465224.

6 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.

7 Soheil Behnezhad, Mahsa Derakhshan, and MohammadTaghi Hajiaghayi. Brief Announcement: Semi-MapReduce Meets Congested Clique. *CoRR*, abs/1802.10297, 2018. arXiv:1802.10297.

8 Andrew Berns, James Hegeman, and Sriram V. Pemmaraju. Super-Fast Distributed Algorithms for Metric Facility Location. In *Automata, Languages, and Programming - 39th International Colloquium, ICALP 2012, Warwick, UK, July 9-13, 2012, Proceedings, Part II*, pages 428–439, 2012. doi:10.1007/978-3-642-31585-5_39.

9 Keren Censor-Hillel, Petteri Kaski, Janne H. Korhonen, Christoph Lenzen, Ami Paz, and Jukka Suomela. Algebraic Methods in the Congested Clique. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 143–152, New York, NY, USA, 2015. ACM. doi:10.1145/2767386.2767414.

10 Moses Charikar, Samir Khuller, David M. Mount, and Giri Narasimhan. Algorithms for Facility Location Problems with Outliers. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '01, pages 642–651, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

11 Jeffrey Dean and Sanjay Ghemawat. MapReduce: A Flexible Data Processing Tool. *Commun. ACM*, 53(1):72–77, January 2010. doi:10.1145/1629175.1629198.

12 Danny Dolev, Christoph Lenzen, and Shir Peled. "Tri, Tri Again": Finding Triangles and Small Subgraphs in a Distributed Setting. In *Proceedings of the 26th International Symposium on Distributed Computing (DISC)*, pages 195–209, 2012.

13 Andrew Drucker, Fabian Kuhn, and Rotem Oshman. The communication complexity of distributed task allocation. In *Proceedings of the 30st ACM Symposium on Principles of Distributed Computing (PODC)*, pages 67–76, 2012.

14 Alina Ene, Sungjin Im, and Benjamin Moseley. Fast Clustering Using MapReduce. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pages 681–689, New York, NY, USA, 2011. ACM. doi:10.1145/2020408.2020515.

15 Kiran Garimella, Gianmarco De Francisci Morales, Aristides Gionis, and Mauro Sozio. Scalable Facility Location for Massive Graphs on Pregel-like Systems. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, CIKM '15, pages 273–282, New York, NY, USA, 2015. ACM. doi:10.1145/2806416.2806508.

16 Joachim Gehweiler, Christiane Lammersen, and Christian Sohler. A Distributed $O(1)$-approximation Algorithm for the Uniform Facility Location Problem. In *Proceedings of the 18th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 237–243, 2006. doi:10.1145/1148109.1148152.

17 Mohsen Ghaffari. Distributed MIS via All-to-All Communication. In *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, pages 141–149, 2017. doi:10.1145/3087801.3087830.

18 Mohsen Ghaffari, Themis Gouleakis, Christian Konrad, Slobodan Mitrovic, and Ronitt Rubinfeld. Improved Massively Parallel Computation Algorithms for MIS, Matching, and Vertex Cover. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom, July 23-27, 2018*, pages 129–138, 2018. doi:10.1145/3212734.3212743.

19 Michael T. Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, Searching, and Simulation in the Mapreduce Framework. In *Proceedings of the 22nd International Conference on Algorithms and Computation*, ISAAC'11, pages 374–383, Berlin, Heidelberg, 2011. Springer-Verlag. doi:10.1007/978-3-642-25591-5_39.

20 James W. Hegeman and Sriram V. Pemmaraju. Lessons from the Congested Clique applied to MapReduce. *Theor. Comput. Sci.*, 608:268–281, 2015. doi:10.1016/j.tcs.2015.09.029.

21 James W. Hegeman and Sriram V. Pemmaraju. Sub-logarithmic distributed algorithms for metric facility location. *Distributed Computing*, 28(5):351–374, 2015. doi:10.1007/s00446-015-0243-x.

22 James W. Hegeman, Sriram V. Pemmaraju, and Vivek Sardeshmukh. Near-Constant-Time Distributed Algorithms on a Congested Clique. *CoRR*, abs/1408.2071, 2014. arXiv:1408.2071.

23 Stephan Holzer and Nathan Pinsker. Approximation of Distances and Shortest Paths in the Broadcast Congest Clique. *arXiv preprint, arXiv:1412.3445*, 2014.

24 Tanmay Inamdar, Shreyas Pai, and Sriram V. Pemmaraju. Large-Scale Distributed Algorithms for Facility Location with Outliers. *CoRR*, abs/1811.06494, November 2018. arXiv:1811.06494.

25 Kamal Jain and Vijay V. Vazirani. Approximation Algorithms for Metric Facility Location and k-Median Problems Using the Primal-dual Schema and Lagrangian Relaxation. *J. ACM*, 48(2):274–296, March 2001. doi:10.1145/375827.375845.

26 Tomasz Jurdziński and Krzysztof Nowicki. MST in O(1) Rounds of Congested Clique. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '18, pages 2620–2632, Philadelphia, PA, USA, 2018. Society for Industrial and Applied Mathematics.

27 Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A Model of Computation for MapReduce. In *Proceedings of the Twenty-first Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, pages 938–948, Philadelphia, PA, USA, 2010. Society for Industrial and Applied Mathematics.

28 Hartmut Klauck, Danupon Nanongkai, Gopal Pandurangan, and Peter Robinson. Distributed Computation of Large-scale Graph Problems. In *Proceedings of the Twenty-sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '15, pages 391–410, Philadelphia, PA, USA, 2015. Society for Industrial and Applied Mathematics.

29 Christoph Lenzen. Optimal Deterministic Routing and Sorting on the Congested Clique. In *Proceedings of the 31st ACM Symposium on Principles of Distributed Computing (PODC)*, pages 42–50, 2013. doi:10.1145/2484239.2501983.

30 Shi Li. A 1.488 Approximation Algorithm for the Uncapacitated Facility Location Problem. In *Proceedings of the 38th International Conference on Automata, Languages and Programming - Volume Part II*, ICALP'11, pages 77–88, Berlin, Heidelberg, 2011. Springer-Verlag.

31 Zvi Lotker, Boaz Patt-Shamir, Elan Pavlov, and David Peleg. Minimum-Weight Spanning Tree Construction in $O(\log \log n)$ Communication Rounds. *SIAM Journal on Computing*, 35(1):120–131, 2005.

32 Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM. doi:10.1145/1807167.1807184.

33 Ramgopal R. Mettu and C. Greg Plaxton. The Online Median Problem. *SIAM J. Comput.*, 32(3):816–832, March 2003. doi:10.1137/S0097539701383443.

34 Danupon Nanongkai. Distributed approximation algorithms for weighted shortest paths. In *Proceedings of the 46th ACM Symposium on Theory of Computing (STOC)*, pages 565–573, 2014.

35 Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. Fast Distributed Algorithms for Connectivity and MST in Large Graphs. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, pages 429–438, New York, NY, USA, 2016. ACM. doi:10.1145/2935764.2935785.

36 Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. On the Distributed Complexity of Large-Scale Graph Computations. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures, SPAA 2018, Vienna, Austria, July 16-18, 2018*, pages 405–414, 2018. doi:10.1145/3210377.3210409.

37 Boaz Patt-Shamir and Marat Teplitsky. The Round Complexity of Distributed Sorting. In *Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 249–256, 2011. doi:10.1145/1993806.1993851.

38 Jonathan A. Silva, Elaine R. Faria, Rodrigo C. Barros, Eduardo R. Hruschka, André C. P. L. F. de Carvalho, and João Gama. Data Stream Clustering: A Survey. *ACM Comput. Surv.*, 46(1):13:1–13:31, July 2013. doi:10.1145/2522968.2522981.

39 Mikkel Thorup. Quick k-Median, k-Center, and Facility Location for Sparse Graphs. *SIAM Journal on Computing*, 34(2):405–432, 2005. doi:10.1137/S0097539701388884.

Equilibria of Games in Networks for Local Tasks

Simon Collet[1]
CNRS and University Paris Diderot, France

Pierre Fraigniaud[2]
CNRS and University Paris Diderot, France

Paolo Penna
ETH Zurich, Switzerland

Abstract

Distributed tasks such as constructing a maximal independent set (MIS) in a network, or properly coloring the nodes or the edges of a network with reasonably few colors, are known to admit efficient distributed randomized algorithms. Those algorithms essentially proceed according to some simple generic rules, by letting each node choosing a temptative value at random, and checking whether this choice is consistent with the choices of the nodes in its vicinity. If this is the case, then the node outputs the chosen value, else it repeats the same process. Although such algorithms are, with high probability, running in a polylogarithmic number of rounds, they are not robust against actions performed by rational but selfish nodes. Indeed, such nodes may prefer specific individual outputs over others, e.g., because the formers suit better with some individual constraints. For instance, a node may prefer not being placed in a MIS as it is not willing to serve as a relay node. Similarly, a node may prefer not being assigned some radio frequencies (i.e., colors) as these frequencies would interfere with other devices running at that node. In this paper, we show that the probability distribution governing the choices of the output values in the generic algorithm can be tuned such that no nodes will rationally deviate from this distribution. More formally, and more generally, we prove that the large class of so-called LCL tasks, including MIS and coloring, admit simple "Luby's style" algorithms where the probability distribution governing the individual choices of the output values forms a Nash equilibrium. In fact, we establish the existence of a stronger form of equilibria, called symmetric trembling-hand perfect equilibria for those games.

2012 ACM Subject Classification Theory of computation → Distributed algorithms, Theory of computation → Algorithmic game theory, Theory of computation → Network games

Keywords and phrases Local distributed computing, Locally checkable labelings

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.6

1 Introduction

1.1 Motivation and Objective

In networks, independent sets and dominating sets can be used as backbones to collect, transfer, and broadcast information, and/or as cluster heads in clustering protocols (see, e.g., [19, 23]). Hence, a node belonging to some selected independent or dominating set may be subject to future costs in term of energy consumption, computational efforts, and bandwidth usage. As a consequence, rational selfish nodes might be tempted to deviate

[1] Funded by the European Research Council (ERC) under the H2020 research and innovation program (grant No 648032).
[2] Additional supports from the ANR project DESCARTES, and from the Inria project GANG.

© Simon Collet, Pierre Fraigniaud, and Paolo Penna;
licensed under Creative Commons License CC-BY

from the instructions of an algorithm used to construct such sets, so that to avoid becoming member of the independent set, or dominating set, under construction. On the other hand, the absence of a backbone, or of cluster heads, may penalize the nodes. Hence every node is subject to a tension between (1) facilitating the *obtention* of a solution, and (2) avoiding certain *forms* of solutions.

A large class of randomized algorithms [5, 22] for constructing maximal independent sets (MIS) proceed in synchronous *rounds*, where a round allows every node to exchange information with its neighbors in the network, and to perform some individual computation. Roughly, at each round of these algorithms, every node i which has not yet decided applies to enter the MIS with a certain probability p_i. If a node applies to enter the MIS, and none of its neighbors simultaneously apply, then the former node enters the MIS, and, subsequently, all its neighbors renounce to enter the MIS. If two adjacent nodes simultaneously apply to enter the MIS, then there is a conflict, and both nodes remain undecided, and go to the next round. The round complexity of the algorithm heavily depends on the choice of the probability p_i that node i applies to enter the MIS, which may typically depend on the degree of node i, and may vary along with the execution of the algorithm, as node i accumulates more and more information about its neighborhood. Hence, a node i aiming at avoiding entering the MIS might be tempted to deviate from the algorithm by setting p_i small. However, if all nodes deviate in this way, then the algorithm may take a very long time before converging to a solution. The same holds whenever all nodes are aiming at entering the MIS.

Similar phenomenons may appear for other problems, like, e.g., coloring [7], that is, an abstraction of frequency assignment in radio networks. For solving this task, typical algorithms provides every node i with a probability distribution $\mathcal{D}_i$ over the colors, and node i chooses color c at random with probability $\mathcal{D}_i(c)$. If this color does not conflict with the chosen colors of its neighbors, then node i adopts this color, else it performs another random choice, and repeats until no conflicts with the neighbors occur. However, some frequencies might be preferred to others because, e.g., some frequencies might be conflicting with local devices present at the node. As a consequence, not all colors are equal for the nodes, and while each node is aiming at constructing a coloring quickly (in order to take benefit from the resulting radio network), it is also aiming at avoiding being assigned a color that it does not want. Therefore, in a random assignments of colors, every node might be tempted to give more weight to its preferred colors than to its non desired colors, and if all nodes deviate in this way, then the algorithm may take a long time before converging to a solution, if converging at all.

In fact, such phenomena as those listed above are susceptible to occur for many network problems, typically for solving so-called *locally checkable labeling* tasks [24], or LCL tasks for short.

1.1.1 Locally Checkable Labelings

LCL tasks [24] form a large class of classical problems, including maximal independent set, coloring, maximal matching, minimal dominating set, etc., studied for more than 20 years in the framework of distributed computing in networks. An LCL task is characterized by a finite set of *labels*, and a set of *good* labeled balls[3] of radius at most t, for some fixed $t \geq 0$.

[3] A *ball* of radius t is a graph with one identified node, called *center*, and with all the other nodes at distance at most t from the center. In a graph G, a ball of radius t centered at some node v is the subgraph induced by all nodes at distance at most t from v in G. A *labeled* ball is a ball whose every node is provided with a label (i.e., a bit-string).

For instance, in the MIS task, balls are of radius 1, a label is either $\bullet$ (interpreted as being member of the independent set), or $\circ$ (interpreted as not being member of the independent set), and a labeled ball is good if either (1) its center is labeled $\bullet$, and all its neighbors are labeled $\circ$, or (2) its center is labeled $\circ$, and at least one of its neighbors is labeled $\bullet$. Similarly, in k-coloring, the labels are in $\{1,\dots,k\}$, balls are of radius 1, and a ball is good if the label of the center is different from the label of each of its neighbors.

Solving an LCL task consists in designing a distributed algorithm resulting in all nodes collectively assigning a label to each of them, such that all resulting balls are good.

In the following, we restrict ourselves to the large class of LCL tasks that are sequentially solvable by a greedy algorithm that (1) picks the nodes one by one in an arbitrary order, and (2) sets the label of each node when picked, after solely consulting the vicinity of the node. For instance, MIS is greedily constructible, as well as $(\Delta+1)$-coloring in networks of maximum degree Δ. Instead, Δ-coloring is not greedily constructible, as witnessed by 2-coloring even cycles. We restrict ourselves to greedily constructible LCL tasks because non greedily constructible tasks are hard to handle in the distributed network computing setting. Indeed, for solving such tasks, far away nodes might be forced to coordinate, yielding poor locality, as witnessed by, again, 2-coloring even cycles (which cannot be solved in less than $\Omega(n)$ rounds [21]).

1.1.2 A Generic "Luby's Style" Randomized Algorithm for LCL Tasks

A generic randomized algorithm for LCL tasks, directly inspired from [5, 22], and therefore often referred to as "Luby's style" algorithm, performs as follows. Every node v aims at computing its label, $\mathit{label}(v)$, for a given LCL task. The labels should be such that all resulting labeled balls are good with respect to the considered task. Node v starts with initial value $\mathit{label}(v)=\bot$. Let t be the radius of the task, i.e., the maximum radius of the labeled balls defining the tasks (both MIS and coloring have radius 1).

Distributed Construction Algorithm: At each round, every node v which has not yet terminated observes the ball of radius t around it in the network (including its structure, and the already fixed nodes' labels). Then v chooses a random *temporary* label, $\mathit{tmp\text{-}label}(v)$, compatible with the current fixed labels of the nodes in the observed ball. Next, v observes the ball of radius t around it in the network again, and recovers the temporary labels randomly chosen by the nodes in the ball. If $\mathit{tmp\text{-}label}(v)$ is compatible with all fixed labels, and with all temporary labels in the ball of radius t centered at v (i.e., if the observed labeled ball in good w.r.t. the considered LCL task), then v sets $\mathit{label}(v)$ as equal to $\mathit{tmp\text{-}label}(v)$. The value of $\mathit{label}(v)$ is then fixed, and it is not subject to any modification in the future. Otherwise, node v goes to the next round.

Note that assuming that the LCL task is greedily constructive prevents nodes from being blocked by nodes that terminated at previous rounds: every node has always at least one label at its disposal for building a good ball around it. (It can be easily checked that, for all greedily constructible LCL tasks, the generic distributed construction algorithm terminates, and outputs correctly, as long as every label has non zero probability to be chosen). Note that each phase of the generic algorithm takes only $2t$ rounds for LCL task of radius t, as every node performs two snapshots of the labels (fixed or temporary) in its t-ball.

The random choice of $\mathit{tmp\text{-}label}(v)$ is governed by some probability distribution $\mathcal{D}$, which is actually characterizing the algorithm, and must be tuned according to the LCL task at hand. Importantly, $\mathcal{D}$ may depend on the round, and may also depend on the structure of

v's neighborhood as observed during the previous rounds. It is known that, for many tasks such as Maximal Independent Set (MIS), and $(\Delta+1)$-coloring, there exist distributions $\mathcal{D}$ enabling the generic distributed construction algorithm to terminate in $O(\log n)$ rounds in n-node networks (see, e.g., [5, 7, 22]). In this paper, we consider the following issue:

What if selfish nodes are not playing according to the desired distribution $\mathcal{D}$?

To address this issue, we define *LCL games*.

1.1.3 LCL Games

To every LCL task can be associated a game, that we call *LCL game*, and that we define as follows. Let G be a connected simple graph. Every node v of G is a rational and selfish *player*, playing the game with the ultimate goal of maximizing its payoff while performing the generic distributed construction algorithm described in Section 1.1.2.

Strategy. A *strategy* for a node v is a probability distribution $\mathcal{D}$ over the labels compatible with the ball of radius t centered at v, which may depend on the history of v during the execution of the generic algorithm. For instance, in the MIS game, a strategy is a probability p to propose itself for entering the MIS. Similarly, in the $(\Delta+1)$-coloring game, a strategy is a distribution of probabilities over the set of remaining colors compatible with the colors already assigned to the neighbors. (This set may even include a "fake" color 0 if nodes do not need to be systematically participating to a choice of color at every round [7]).

▶ Remark. The distribution $\mathcal{D}$ over the labels compatible with the ball of radius t centered at v is the unique item subject to non-orthodox behaviors. In particular, in LCL games, every node executes the prescribed algorithm, forwards messages correctly, and does not lie about its internal state, apart from what is concerning its private strategy for choosing its temporary label at random.

The *strategy* of a node at a round, i.e., the distribution $\mathcal{D}$ of probability over the labels, may depend on the history of that node at any point in time during the execution of the generic distributed construction algorithm. On the other hand, the individual strategies depend only on the knowledge accumulated by the nodes along with the execution of the algorithm. In fact, at the beginning of the algorithm, player v does not even know which node she will play in the network, and just knows that the network may belong to some given graph family (like, e.g., cycles, planar graphs, etc.).

Payoff. The *payoff* of the nodes is aiming at capturing the tension between the objective of every node to compute a global solution rapidly (as this global solution brings benefits to every node), versus avoiding certain forms of solutions (which may not be desirable from an individual perspective). We denote by $pref_v$ a preference function, which is an abstraction of how much node v will "suffer" in the future according to a computed solution. For instance, in the MIS game where nodes do not want to belong the constructed MIS, one could set

$$pref_v(I) = \begin{cases} 0 & \text{if } v \in I \\ 1 & \text{otherwise} \end{cases}$$

for every MIS I. More specifically, we define, for each node v,

$$pref_v : \{\text{good balls}\} \to [0,1]$$

by associating to each good ball B centered at v the preference $pref_v(B)$ of v for that ball. The *payoff function* π_v of node v at the completion of the algorithm is decaying with the number k of rounds before the algorithm terminates at v. More precisely, we set

$$\pi_v = \delta^k \; pref_v(B_v)$$

where $0 < \delta < 1$ is a discount factor, B_v is the good ball centered at v as returned by the algorithm, and k is the number of rounds performed before all nodes in B_v fix their labels.

The choice of $\delta \in (0, 1)$ reflects the tradeoff between the quality of the solution from the nodes' perspective, and their desire to construct a global solution quickly. Note that k is at least the time it takes for v to fix its label, and at most the time it takes for the algorithm to terminate at all nodes. If the algorithm does not terminate around v, that is, if a label remains perpetually undecided in at least one node of B_v, then we set $\pi_v = 0$.

- The payoff of a node v will thus be large if all nodes in B_v decide quickly (i.e., k is small), and if the labels computed in B_v suits node v (i.e., $pref_v(B_v)$ is close to 1).
- Conversely, if v or another node in its ball B_v takes many rounds before deciding a label (i.e., k is large), or if node v is not satisfied by the computed solution in B_v (i.e., $pref_v(B_v)$ is close to 0), then the payoff of v will be small.

In particular, if the preference for every good ball is the same, then maximizing the payoff is equivalent to completing the task as quickly as possible. Instead, if the preference is very small for some balls, then nodes might be willing to slow down the completion of the task, with the objective of avoiding being the center of such a ball, in order to maximize their payoff. That is, such nodes may bias their distribution $\mathcal{D}$ towards preferred good balls, even if this is at the price of increasing the probability of conflicting with the choices of close nodes, resulting in more iterations before reaching convergence.

1.2 Our Results

We show that LCL games have *trembling-hand perfect equilibria*, that is, a stronger form of sequential equilibria due to Reinhard Selten [29], which are themselves a stronger form of Nash equilibria. Trembling-hand perfect equilibria include the possibility of off-the-equilibrium play, i.e., players may, with small probabilities, choose unintended strategies. In contrast, in Nash equilibria, players are assumed to play precisely as specified by the equilibrium. We show the following:

▶ **Theorem 1.** *For any greedily constructible locally checkable labeling, the LCL game associated to that labeling has a symmetric trembling-hand perfect equilibrium.*

Therefore, in particular, for many tasks occurring in the context of distributed network computing such as MIS, and $(\Delta + 1)$-coloring, there exist strategies played by the nodes of the network for solving these tasks such that no nodes have incentive to deviate from these strategies. Moreover, the related equilibria are strong forms of Nash equilibria which ensure that the players behave rationally even off the equilibrium path.

To establish Theorem 1, we first notice that LCL games belongs to the class of *extensive* games with *imperfect information*, because a node plays arbitrarily many times, and is not necessarily aware of the actions taken by far away nodes in the network. Also, LCL games belongs to the class of games with *infinite horizon* and *finite action set*: the horizon is infinite because neighboring nodes may perpetually prevent each other from terminating, and the action set is supposed to be finite (as long as the set of labels is finite). However, the classical game theoretical result [12] does not explicitly apply to LCL games. Indeed, first, in LCL games the actions of far-away nodes are not observable. Second, the imperfect information in [12] is solely related to the fact that players play simultaneously, while, again,

in LCL games, imperfect information also refers to the fact that each node is not aware of the states of far away nodes in the network. It follows that the first step in our proof consists of revisiting the results in [12] for extending them, as specified in the following result.

▶ **Lemma 2.** *Every infinite, continuous, measurable, well-rounded, extensive game with perfect recall and finite action set has a trembling-hand perfect equilibrium. Moreover, if the game is, in addition, symmetric, then it has a symmetric trembling-hand perfect equilibrium.*

The hypotheses regarding the nature of the strategy, and the nature of the payoff function (continuity, measurability, etc.) are standard in the framework of extensive games. The notion of *well-rounded* game is new, and is used to capture the fact that the nodes play in synchronous rounds in LCL games. The fact that the equilibrium is symmetric is crucial as far as games in networks are concerned since, in LCL games, as in randomized distributed computing in general, the instructions given to all nodes are identical, and the behavior of the nodes only vary along with the execution of the algorithm when they progressively discover their environment. Extending the results in [12] is quite technical, but follows the standard methods for establishing such results in game theory. Therefore, we have chosen not to include the proof of Lemma 2 in this extended abstract.

The more interesting part of the proof, as far as local distributed computing in networks is concerned, is to show that LCL games satisfy all requirements stated in Lemma 2. This is the role of the following result:

▶ **Lemma 3.** *LCL games are symmetric, infinite, continuous, measurable, well-rounded, extensive games with perfect recall and finite action set.*

Lemmas 2 and 3 together prove Theorem 1. The rest of the paper is therefore focussing on formalizing LCL games, and on proving Lemma 3.

1.3 Related Work

Let us first position our result into the various settings of game theory. Indeed, games take various forms, and the types of equilibria that can be satisfied by these games vary according to their forms. Table 1 surveys the results regarding equilibria for various game settings, from the finite strategic games to the extensive games with imperfect information (we restrict our attention to games with a finite number of players). Recall that *trembling-hand perfect* equilibria [29] are refinements of *sequential* equilibria, which are themselves refinements of *subgame-perfect* equilibria, all of them being Nash equilibria. In Table 1, we distinguish strategic games (i.e., 1-step games like, e.g., prisoner's dilemma) from extensive games (i.e., game trees with payoffs, like, e.g., monetary policy in economy). For the latter class, we also distinguish games with perfect information (i.e., every player knows exactly what has taken place earlier in the game), from the games with imperfect information. We also distinguish finite games (i.e., games with a finite number of pure strategies, and finite number of repetitions) from games with infinite horizon (i.e., games which can be repeated infinitely often). The latter class of games is also split into games with finite numbers of actions, and games with infinite set of actions (like, e.g., when fixing the price of a product). In particular, Fudenberg and Levine [12] have proved that, under specific assumptions, every extensive game with imperfect information and finite action set has a sequential equilibrium (for behavior strategies). The specific class of games for which this result holds can be described as extensive games with observable actions, simultaneous moves, perfect recall, and finite action set, plus some continuity requirements. Although this class of games captures repeated games, and contains natural games in economy, LCL games are not explicitly included into this class. Indeed, as we already mentioned, the actions of far-away nodes are not observable

Table 1 A summary of results about the existence of equilibria.

<table>
<tr><th></th><th>Strategic Games</th><th>Extensive games with perfect information</th><th>Extensive games with imperfect information</th></tr>
<tr><td>Finite games</td><td rowspan="2">[25]
Nash equilibrium
Mixed strategies</td><td>[28]
Subgame-perfect equilibrium
Pure strategies</td><td>[29]
Trembling-hand perfect eq.
Behavior strategies</td></tr>
<tr><td>Games with a finite action set</td><td>[12]
Subgame-perfect equilibrium
Pure strategies</td><td>[12]
Sequential equilibrium
Behavior strategies</td></tr>
<tr><td>Games with an infinite action set</td><td>[11]
[14]
Nash equilibrium
Mixed strategies</td><td>[16]
Subgame-perfect equilibrium
Pure strategies</td><td>[10]
Nash equilibrium
Behavior strategies</td></tr>
</table>

in LCL games, and, in these latter games, imperfect information also refers to the fact that each node is not aware of the states of far away nodes in the network.

We now list some previous works related to games in networks (for network formation games, see, e.g., [6, 17]). Many games in networks have complete information, and, among games with incomplete information, a large part of the literature is dedicated to single-stage games where players are not initially aware of the network topology (see the survey [18]). Repeated games in networks have also been considered a lot in the literature (again, see [18]). These games differ from LCL games since, in repeated games, the utility of a player depends on each round, and it is computed pairwise with each neighbor, while, in LCL games, the utility is computed solely when the player terminates, and may depend on the whole neighborhood. Regarding games with incomplete information involving communications in networks, it is worth mentioning [2, 8, 9, 13, 15]. However, all these work mostly refer to games in which the players' actions consist in choosing which information to reveal, and to whom it should be revealed. Instead, in LCL games, players actions are always fully observable by their neighbors at distance $\leq t$, where t is the maximum radius of the good balls for the considered LCL task.

Probably the first contribution to distributed computing by rational agents is [1], which studies leader election in various networks, including complete networks and rings. Different forms of Nash equilibria are shown to exist, for both synchronous and asynchronous computing. The contribution in [3] extended and generalized the results in [1] by considering other problems (consensus, renaming, etc.), and by identifying different utility functions that encompass different preferences of players in a distributed system: communication preference, solution preference, and output preference. The paper [4] carried on this line of research, by enlarging the considered set of problems to coloring, partition, orientation, etc., and by addressing the question of how much global information agents should know a priori about the network in order for equilibria to exist. All these previous work differ from our approach in many ways. First, in [1, 3, 4], the agents strategies define the algorithm itself, including which messages to send, which information to reveal, etc. Instead, in this paper, the agents strategies solely consist in choosing a probability distribution on the possible outputs (at each round, depending on the history of the player). Second, the algorithms in the three aforementioned papers are "global" in the sense that they can take $\Omega(n)$ rounds in n-node networks. Instead, our (generic) algorithm is in essence "local", i.e., it is expected to converge in a polylogarithmic number of rounds, even in networks with large diameter. Last but not least, we consider a whole family of tasks at once (all "reasonable" LCL tasks) while the three aforementioned papers address each task separately, each one with its own algorithm.

2 The Extensive Games Related to LCL Games

In this section, we specify the type of games we are interested in, aiming at capturing the characteristics of LCL games. We focus on extensive games with imperfect information, and we include infinite horizon in the analysis of such games. We formally define all the concepts appearing in the statement of Lemma 2, and/or useful for formally defining LCL games, and proving Lemma 3. In particular, we define the novel notion of *well-rounded* games, which fits with distributed network computing in the LOCAL model [26].

2.1 Basic Definitions

Recall that an extensive game is a tuple $\Gamma = (N, A, X, P, U, p, \pi)$, where:

- $N = \{1, \dots, n\}$ is the set representing the *players* of the game. An additional player, denoted by c, and called *chance*, is modeling all external random effects that might occur in the course of the game.
- A is the (finite) *action set*, i.e., a finite set representing the actions that can be made by each player when she has to play.
- X is the *game tree*, that is, a subset of $A^* \cup A^\omega$ where A^* (resp., A^ω) denotes the set of finite (resp., infinite) strings with elements in A, satisfying the following properties:
  - the empty sequence $\varnothing \in X$;
  - X is stable by prefix;
  - if $(a_i)_{i=1,\dots,k} \in X$ for every $k \geq 1$, then $(a_i)_{i\geq 1} \in X$.

  The set X is partially ordered by the prefix relation, denoted by $\preceq$, where $x \preceq y$ means that x is a prefix of y, and $x \prec y$ means that x is a prefix of y distinct from y. The elements of X are called *histories*. A history x is *terminal* if it is a prefix of no other histories in X. In particular, every infinite history in X is terminal. The set of terminal histories is denoted by Z. If the longest history is finite then the game has *finite horizon*, otherwise it has *infinite horizon*. For every non-terminal history x, we denote by $A(x) = \{a \in A : (x, a) \in X\}$ the set of available actions after the history x.
- P is the *player partition*, i.e., a function $P : X \setminus Z \to N \cup \{c\}$ that assigns a player to each non-terminal history. $P(x)$ is the player who has to play after the history x. The sets $P_i = \{x \in X \setminus Z : P(x) = i\}$, for $i \in N \cup \{c\}$, called *player sets*, form a partition of $X \setminus Z$.
- U is the *information partition*, that is, a refinement of the player partition, whose elements are called *information sets*, such that for every $u \in U$, and for every two histories x, y in this *information set* u, we have $A(x) = A(y)$, i.e., the sets of available actions after x and after y are identical. We can therefore define $A(u)$ as the set of actions available after the information set u. Formally, $A(u) = \{a \in A : (x, a) \in X \text{ for every } x \in u\}$. For every history x, the information set containing x is denoted by $u(x)$. We also define $P(u)$ as the player who has to play after the information set u has been reached, and for every player i, the set $U_i = \{u \in U : P(u) = i\}$. The collection $\{U_i, i \in N \cup \{c\}\}$ forms a partition of U. Information sets regroup histories that are indistinguishable to players. Since the chance player c is not expected to behave rationally, we will simply put $U_c = \{\{x\}, x \in P_c\}$.
- p is a function that assigns to every history x in P_c (the player set of the chance c) a probability distribution over the set $A(x)$ of available actions after the history x. This *chance function* p is supposed to be common knowledge among the players.

- π is the *payoff function*, that is, $\pi : Z \to \mathbb{R}^n$ assigns the payoff (a real value) to every player in N for every terminal history of the game. We assume that every payoff is in $[-M, M]$ for some $M \geq 0$.

2.2 Well-Rounded Games

We introduce the concept of *rounds* in extensive games, and of *well-rounded* games.

▶ **Definition 4.** The *round function* r of an extensive game assigns a positive integer to every non terminal history x, defined by $r(x) = |Rec(x)|$ where

$$Rec(x) = \{x' \in X \mid x' \prec x \text{ and } P(x') = P(x)\}.$$

We call $r(x)$ the *round* of x. The round of a finite terminal history is the round of its predecessor, and the round of an infinite history z is $r(z) = \infty$. An extensive game Γ for which the round function is non decreasing with respect to the prefix relation, i.e.,

$$y \preceq x \Longrightarrow r(y) \leq r(x),$$

is said to be *well-rounded*.

Note that not every game is well-rounded, because two histories x and y such that $x \preceq y$ do not necessarily satisfy $P(x) = P(y)$. In a well-rounded game, since r is non decreasing, we have that, for any non terminal history x, every player has played at most $r(x) + 1$ times before x. Moreover, every player which has played less than $r(x)$ times before x will never play again after x.

Let $u \in U_i$ and $u' \in U_i$ be two (non necessarily distinct) information sets of the same player i, for which there exist $x \in u$, $x' \in u'$, and $a \in A(u')$, such that $(x', a) \preceq x$. Recall that an extensive game is said to have *perfect recall* if, for every such i, u, u' and a, we have:

$$\forall y \in u, \exists y' \in u' \mid (y', a) \preceq y.$$

The following lemma will allow us to safely talk about the round of an information set.

▶ **Lemma 5.** *Let Γ be an extensive game with perfect recall, and let $x \in X$ and $x' \in X$ be two non terminal histories in the same information set $u \in U$. Then x and x' have the same round.*

Proof. We first observe the following. Let Γ be an extensive game with perfect recall, and let $y \in X$ be a finite history. Let $y' \in X$ and $y'' \in X$ for which there exists $u \in U$ such that

$$y' \in Rec(y) \cap u \text{ and } y'' \in Rec(y) \cap u.$$

Then $y' = y''$. Indeed, since both y' and y'' are in $Rec(y)$, we have that both are prefixes of y, and thus one of the two is a prefix of the other. Assume, w.l.o.g., that $y'' \prec y' \prec y$ (as, if $y' = y''$ then we are done). Let a be the action such that $(y'', a) \preceq y'$. Since the game has perfect recall, there must exist a history $y''' \in u$ such that $(y''', a) \preceq y''$. Thus $y''' \prec y'' \prec y' \prec y$.We can repeat the same reasoning for y''' and y'' as we did for y'' and y'. In this way, we construct an infinite strictly decreasing sequence of histories, which contradicts the fact that y is finite.

If both $Rec(x)$ and $Rec(x')$ are empty, then x and x' have the same round. Assume, w.l.o.g., that $Rec(x) \neq \varnothing$, and let $y \in Rec(x)$. Let a be the action such that $(y, a) \preceq x$. Since the game has perfect recall, there exists $y' \in u(x')$ such that $(y', a) \preceq x'$. Therefore

$y' \prec x'$ and $P(y') = P(y) = P(x) = P(x')$. It follows that $y' \in Rec(x')$. Thus, for any $y \in Rec(x)$, we have identified a corresponding $y' \in Rec(x')$. This mapping from $Rec(x)$ to $Rec(x')$ is one-to-one. Indeed, let y_1 and y_2 in $Rec(x)$, and let y'_1 and y'_2 in $Rec(x')$ be the corresponding histories. If $y'_1 = y'_2$, then, since $u(y_1) = u(y'_1)$ and $u(y_2) = u(y'_2) = u(y'_1)$, we get that

$$y_1 \in Rec(x) \cap u(y'_1) \text{ and } y_2 \in Rec(x) \cap u(y'_1).$$

It follows from the above observation that $y_1 = y_2$. Thus the mapping is one-to-one, and hence $r(x) \leq r(x')$. It follows that we also have $Rec(x') \neq \varnothing$. Therefore, we can apply the same reasoning by switching the roles of x and x', which yields $r(x') \leq r(x)$. Thus $r(x) = r(x')$. ◀

2.3 Strategies, Outcomes, and Expected Payoff

In this section, we first recall several basic concepts about extensive games with perfect recall. Without loss of generality, we restrict our attention to *behavioral strategies* since such strategies are outcome-equivalent to mixed strategies thanks to [20]. The main objective of this section is to define the *expected payoff function*, which is novel as it is adapted to infinite games.

Recall that, for an information set u, the *local strategy* $b_{i,u}$ of a player i is a probability distribution over the set $A(u)$ of actions available given u. The set of local strategies of player i for u is denoted by $B_{i,u}$. The *behavioral strategy* b_i of a player i is a function which assigns a local strategy $b_{i,u}$ to every information set u of this player. The set of all behavioral strategies of player i is denoted by B_i. A *strategy profile* is a n-tuple of behavioral strategies, one for each player. The set of all strategy profiles is $B = \times_{i \in N} B_i$. For each player i, we denote by B_{-i} the set $\times_{j \neq i} B_j$. Since

$$B = B_i \times B_{-i} = \times_{i \in N} B_i,$$

a strategy profile b can be identified different ways, as $b = (b_i, b_{-i}) = (b_1, b_2, \dots, b_n)$. If every player plays according to a strategy profile b, then the outcome of the game in entirely determined, in the sense that every history x has a probability $\rho_b(x)$ of being reached. For every strategy profile b, and every history $x = (a_i)_{i=1,\dots,k}$ where $k \in \mathbb{N} \cup \infty$, the *realization probability* of x is defined by $\rho_b(\varnothing) = 1$, and

$$\rho_b(x) = \prod_{i=0}^{k-1} b_{P(x_i),u(x_i)}(a_{i+1})$$

where $x_0 = \varnothing$, and, for every positive $i \leq k$, $x_i = (a_1, a_2, \dots, a_i)$. For the chance player c, we simply identify its strategy with the chance function p:

$$P(x_i) = c \Rightarrow b_{P(x_i),u(x_i)}(a_{i+1}) = p(x_i, a_{i+1}).$$

The function $\rho_b : X \to [0, 1]$ is called the *outcome* of the game under the strategy profile b. An outcome $\rho : X \to [0, 1]$ is *feasible* if and only if there exists a strategy profile b such that $\rho = \rho_b$. The set of feasible outcomes of Γ is denoted by O.

We are now ready to define the *expected payoff*. Note that, in a game with infinite horizon, there can be uncountably many terminal histories. Therefore the definition of the probability measure on Z requires some care. For any finite history x, let

$$Z_x = \{z \in Z \mid x \preceq z\}.$$

Note that Z_x might be uncountable. Let Σ be the σ-algebra on Z generated by all sets of the form Z_x for some finite history x. For each strategy profile b, the measure μ_b on Σ is defined by: for every set Z_x, $\mu_b(Z_x) = \rho_b(x)$. This definition ensures that μ_b is a probability measure because $\mu_b(Z) = \mu_b(Z_\varnothing) = \rho_b(\varnothing) = 1$.

▶ **Definition 6.** Let π be a payoff function that is measurable on Σ. The *expected payoff function* Π assigns a real value $\Pi(b)$ to every strategy profile $b \in B$, defined by

$$\Pi(b) = \int_\Sigma \pi \; d\mu_b \; .$$

Note that each component of the expected payoff function is bounded by M, where M is the upper bound on every payoff. A game Γ whose payoff function π is measurable on Σ is said to be a *measurable game*. In the following, we always assume that the considered games are measurable.

2.4 Equilibria, and Subgame Perfection

We now show how to adapt the standard notion of ϵ-equilibria (cf., e.g., [27]) to infinite games (Nash equilibria are special cases of ϵ-equilibria, with $\epsilon = 0$). Recall that a strategy profile b is a ϵ-equilibrium if and only if, for every player i, and every behavior strategy $b'_i \in B_i$ of this player, we have $\Pi_i(b'_i, b_{-i}) - \Pi_i(b) \leq \epsilon$. Similarly, we recall the notions of *subgames* and *subgame perfect equilibria* (see, e.g., [28]). A subtree X' of X is said to be *regular* if no information sets contain both a history in X' and a history not in X'. To each regular subtree X' is associated a game $\Gamma' = (N, A, X', P', U', p', \pi')$, where P', U', p' and π' are the restrictions of P, U, p and π to X', called a *subgame*. The notions of outcomes and expected payoff functions for subgames follow naturally.

▶ **Definition 7.** A strategy profile b is a *subgame perfect ϵ-equilibrium* of an infinite game Γ if and only if, for every subgame Γ', the restriction of b to Γ' is an ϵ-equilibrium.

Note that a subgame perfect ϵ-equilibrium of Γ is an ϵ-equilibrium.

2.5 Metrics

In this section, we now define specific metrics on the set O of feasible outcomes, and on the set of behavior strategy profiles. These definitions are inspired from [12], with adaptations to fit our infinite setting.

▶ **Definition 8.** Let $\rho^1, \rho^2 \in O$ be two feasible outcomes of the same extensive game Γ. We define the following metric d on O: $d(\rho^1, \rho^2) = \sup_{x \in X, x \text{ finite}} 2^{-r(x)} \cdot \left|\rho^1(x) - \rho^2(x)\right|$ where $r(x)$ is the round of the finite history x.

▶ **Lemma 9.** *The function $d : O \times O \to \mathbb{R}$ specified in Definition 8 is a metric.*

Proof. We first show that d satisfies the triangle inequality. Let ρ^1, ρ^2 and ρ^3 be three feasible outcomes of Γ. For any finite history x we have the following:

$$\begin{array}{rll}
 & |\rho^1(x) - \rho^3(x)| & \leq |\rho^1(x) - \rho^2(x)| + |\rho^2(x) - \rho^3(x)| \\
\Rightarrow & 2^{-r(x)}|\rho^1(x) - \rho^3(x)| & \leq 2^{-r(x)}|\rho^1(x) - \rho^2(x)| + 2^{-r(x)}|\rho^2(x) - \rho^3(x)| \\
\Rightarrow & \sup\limits_{\substack{x\in X\\ x \text{ finite}}} 2^{-r(x)}|\rho^1(x) - \rho^3(x)| & \leq \sup\limits_{\substack{x\in X\\ x \text{ finite}}} \left(2^{-r(x)}|\rho^1(x) - \rho^2(x)| + 2^{-r(x)}|\rho^2(x) - \rho^3(x)|\right) \\
\Rightarrow & \sup\limits_{\substack{x\in X\\ x \text{ finite}}} 2^{-r(x)}|\rho^1(x) - \rho^3(x)| & \leq \sup\limits_{\substack{x\in X\\ x \text{ finite}}} 2^{-r(x)}|\rho^1(x) - \rho^2(x)| + \sup\limits_{\substack{x\in X\\ x \text{ finite}}} 2^{-r(x)}|\rho^2(x) - \rho^3(x)| \\
\Rightarrow & d(\rho^1, \rho^3) & \leq d(\rho^1, \rho^2) + d(\rho^2, \rho^3).
\end{array}$$

Next, we prove that d separates different outcomes. Let ρ^1 and ρ^2 be two feasible outcomes such that $d(\rho^1, \rho^2) = 0$. By definition, this implies that, for every finite history x, we have $\rho^1(x) = \rho^2(x)$. Let b^1 and b^2 be two strategy profiles such that $\rho^1 = \rho_{b^1}$ and $\rho^2 = \rho_{b^2}$. Let $x = (a_k)_{k\geq 1}$ be an infinite history, and, for $k \geq 1$, let $x_k = (a_1, a_2, \dots, a_k)$ be the corresponding increasing sequence of its prefixes. By definition, $\rho_{b^1}(x)$ is the limit, for $k \to \infty$, of the sequence $\rho_{b^1}(x_k)$, and $\rho_{b^2}(x_k) \to \rho_{b^2}(x)$ when $k \to \infty$ as well. Since the two sequences are equal, they have the same limit, and therefore $\rho^1(x) = \rho^2(x)$. Since this equality holds for every infinite history x, it follows that $\rho^1 = \rho^2$. ◀

We can use d to define a metric on behavioral strategy profiles as follows. Let $b^1, b^2 \in B$ be two behavioral strategy profiles of the same game Γ. We define the metric d on B by:

$$d(b^1, b^2) = \max\left\{ d\left(\rho_{b^1}, \rho_{b^2}\right),\ \sup_{\substack{i\in N\\ b_i\in B_i}} d\left(\rho_{(b_i, b^1_{-i})}, \rho_{(b_i, b^2_{-i})}\right) \right\}.$$

Finally, we define the continuity of the expected payoff function using the sup norm over $\mathbb{R}^n$. Specifically, the expected payoff function Π is *continuous* if, for every sequence of strategy profiles $(b^k)_{k\geq 1}$, and every strategy profile b, we have:

$$d(b^k, b) \underset{k\to\infty}{\to} 0 \implies \sup_{i\in N} \left|\Pi_i(b^k) - \Pi_i(b)\right| \underset{k\to\infty}{\to} 0.$$

An extensive game Γ is continuous if its expected payoff function is continuous.

2.6 Equilibria of Extensive Games Related to LCL Games

As stated in Lemma 2, it can be proved that every (symmetric) infinite, continuous, measurable, well-rounded, extensive game with perfect recall and finite action set has a (symmetric) trembling-hand perfect equilibrium. (Due to lack of space, this proof is not included in this extended abstract).

3 Proof of Lemma 3

In this section, we show that LCL games satisfy all hypotheses of Lemma 2, from which we derive Theorem 1. We start by formally defining LCL games.

3.1 Formal Definition of LCL Games

Let A be a finite alphabet, $\mathcal{F}$ a family of graphs with at most n vertices, $\mathcal{D}$ a probability distribution over $\mathcal{F}$, and $\mathcal{L}$ a greedily constructible LCL task over graphs in $\mathcal{F}$. Let $good(\mathcal{L})$ be the set of good balls in $\mathcal{L}$, and let t be the radius of $\mathcal{L}$, that is, the largest radius of the good balls. Let $pref_i : good(\mathcal{L}) \mapsto [0, 1]$, be a preference function of player i over good balls, and let $\delta \in (0, 1)$, called *discounting factor*. We define the game

$$\Gamma(\mathcal{L}, \mathcal{D}, pref, \delta) = (N, A, X, P, U, p, \pi)$$

associated to the LCL task $\mathcal{L}$, the distribution $\mathcal{D}$, the preference function $pref = (pref_i)_{i\in N}$, and the discounting factor δ, as follows.

- The player set is $N = \{1, \dots, n\}$.
- The action set is $A \cup \mathcal{F}$ where the actions in $\mathcal{F}$ are only used by the chance player c in the initial move, and the actions in A are used by the actual players in N.

- The first move of the game is made by the chance player, that is, $P(\varnothing) = c$. As a result, a graph $G \in \mathcal{F}$ is selected at random according to the probability distribution $\mathcal{D}$, and a one-to-one mapping of the players to the nodes of G is chosen uniformly at random. From now on, the players are identified with the vertices of the graph G, labeled from 1 to n. Note that $\mathcal{F}$ might be reduced to a single graph, e.g., $\mathcal{F} = \{C_n\}$, and the chance player just selects, for each vertex v, which player $i \in N$ is playing at v (in a one-to-one manner).
- The game is then divided into rounds (corresponding to the intuitive meaning in synchronous distributed algorithms). At each round, the *active* players play in increasing order, from 1 to n. At round 0 every player is active and plays, and every action in A is available.
- At the end of each round (i.e., after every active player has played the same number of times), some players might become *inactive*, depending on the actions chosen during the previous rounds. For every $i \in N$, let $s(i)$ denote the last action played by player i, which we call the *state* of i, and let $ball(i)$ denote the ball of radius t centered at node i. Every player i such that $ball(i) \in good(\mathcal{L})$ at the end of a round becomes *inactive*.
- In subsequent rounds, the set of available actions might be restricted. For every round $r > 0$, and for every active player i, an action $a \in A$ is available to player i if and only if there exists a ball $b \in good(\mathcal{L})$ compatible with the states of inactive players in which $s(i) = a$.
- A history is terminal if and only if either it is infinite, or it comes after the end of a round with every player being inactive after that round.
- Let x be a history. We denote by $actions_i(x)$ the sequence of actions extracted from x by selecting all actions taken by player i during rounds before $r(x)$. (The action possibly made by player i at round $r(x)$, and actions made by a player $j \neq i$ are not included in $actions_i(x)$).
- Let x and y be two non terminal histories such that $P(x) = P(y) = i$. Then x and y are in the same information set if and only if, for every $j \in ball(i)$, we have

  $$actions_j(x) = actions_j(y).$$

  This can be interpreted by the fact that a player i “knows” every action previously taken by any player at distance at most t from i in the graph.
- Let i be a player, and let z be a terminal history. We define the *terminating time* of player i in history z by $time_i(z) = \max\{|actions_j(z)|, j \in ball(i)\} - 1$. The payoff function π of the game is then defined as follows. For every player i, and every terminal history z, we have $\pi_i(z) = \delta^{time_i(z)} \cdot pref_i(ball(i))$. And $\pi_i(z) = 0$ if $time_i(z) = \infty$.

3.2 The proof of Lemma 3

We survey the properties of LCL games, with emphasis on those listed as pre-conditions in the statement of Lemma 2.

▶ **Lemma 10.** *LCL games are well-rounded.*

Proof. This follows directly from the fact that, in a LCL game, (1) every active player plays at every round until it becomes inactive, and (2) once inactive, a player cannot become active again. ◀

▶ **Lemma 11.** *LCL games are symmetric.*

Proof. This follows directly from the fact that, in a LCL game, the position of every player in the actual graph (which might be fixed, or chosen at random in some given family of graphs according to some given distribution) is chosen uniformly at random. ◀

▶ **Lemma 12.** *LCL games have perfect recall.*

Proof. Let $\Gamma = (\mathcal{L}, \mathcal{D}, \mathit{pref}, \delta) = (N, A, X, P, U, p, \pi)$ be an LCL game. Let u and u' be two information sets of the same player i, for which there exists $x \in u$, $x' \in u'$, and $a \in A(u')$ such that $(x', a) \preceq x$. Let y be a history in u. Since x and y are in the same information set u, it follows that, for every player $j \in \mathit{ball}(i)$, we have $\mathit{actions}_j(x) = \mathit{actions}_j(y)$. In particular, this implies that x and y are in the same round. Let y' be the unique history which is a prefix of y with $P(y') = i$, and with $r(y') = r(x')$. (Such a history exists because $r(x') < r(x)$, and $r(y) = r(x)$). Since the players play in the same order at every round, we get that, for every player $j \in \mathit{ball}(i)$, $\mathit{actions}_j(x') = \mathit{actions}_j(y')$. As a consequence, we have $y' \in u'$. Furthermore, since $\mathit{actions}_i(x) = \mathit{actions}_i(y)$, the action played by i after y' must be a, which implies $(y', a) \preceq y$, and concludes the proof. ◀

▶ **Lemma 13.** *The payoff function π of a LCL game is measurable on the σ-algebra Σ corresponding to the game.*

Proof. We prove that, for every player i, and for every $a \in \mathbb{R}$, $\pi_i^{-1}(]a, +\infty[) \in \Sigma$, which implies that π is measurable on Σ. In LCL game, we have $\pi_i : Z \mapsto [0, 1]$. For every $a < 0$, we have $\pi_i^{-1}(]a, +\infty[) = Z \in \Sigma$. Similarly, for every $a > 1$, we have $\pi_i^{-1}(]a, +\infty[) = \varnothing \in \Sigma$. So, let us assume that $a \in]0, 1]$, and let z be a terminal history such that $\pi_i(z) > a$. We have $\mathit{time}_i(z) < \ln a / \ln \delta$, i.e., every player in $\mathit{ball}(i)$ has played only a finite number of times in the history z. Let x be the longest history such that $x \preceq z$, and $r(x) = \mathit{time}_i(z)$. By this setting, the history x' that comes right after x in z is the shortest prefix of z satisfying that every player in $\mathit{ball}(i)$ is inactive. Let z' be a terminal history such that $x' \preceq z'$. Since every player $j \in \mathit{ball}(i)$ is inactive after x', it follows that the state of any such player in z' is the same as its state in z, and thus $\pi_i(z') = \pi_i(z)$. It follows from the above that, for any terminal history z such that $\pi_i(z) > a$, there exists a finite history x' in round $\mathit{time}_i(z) + 1$ such that $z \in Z_{x'} \subseteq \pi_i^{-1}(]a, +\infty[)$. Since there are finitely many histories in round $\mathit{time}_i(z)$, we get that $\pi_i^{-1}(]a, +\infty[)$ is the union of a finite number of sets of the form $Z_{x'}$. As a consequence, it is measurable in Σ. It remains to prove that $\pi_i^{-1}(]0, +\infty[) \in \Sigma$. This simply follows from the fact that

$$\pi_i^{-1}(]0, +\infty[) = \bigcup_{k \geq 1} \pi_i^{-1}(]\frac{1}{k}, +\infty[),$$

and from the fact that Σ is stable by countable unions. ◀

▶ **Lemma 14.** *LCL games are continuous.*

Proof. Let b be a strategy profile, and let $(b^k)_{k \geq 0}$ be a sequence of strategy profiles such that $d(b^k, b) \to 0$ when $k \to \infty$. By definition of the metric d on B (cf. subsection 2.5), we have that $d(\rho_{b^k}, \rho_b) \to 0$ when $k \to \infty$. By definition of the metric on O, we have that, for any finite history x, $|\rho_{b^k}(x) - \rho_b(x)| \underset{k \to \infty}{\longrightarrow} 0$. It follows that, for any set of the form Z_x as defined in subsection 2.3, $|\mu_{b^k}(Z_x) - \mu_b(Z_x)| \underset{k \to \infty}{\longrightarrow} 0$. In other words the sequence of measures μ_{b^k} *strongly converges* to μ_b. Since, for every player i, the function π_i is measurable and bounded, it follows that

$$\int_\Sigma \pi_i \; d\mu_{b^k} \underset{k \to \infty}{\longrightarrow} \int_\Sigma \pi_i \; d\mu_b.$$

Therefore, $\Pi_i(b^k) \underset{k\to\infty}{\longrightarrow} \Pi_i(b)$, and thus the expected payoff function Π is continuous. ◀

Lemmas 10-14 show that every LCL game satisfies the requirements of Lemma 2, that is, every LCL game satisfies Lemma 3. ◀

4 Conclusion and Further Work

In this paper, we have proved that natural games occurring in the framework of local distributed network computing have trembling-hand perfect equilibria, a strong form of Nash equilibria. Further study includes the analysis of the performances of the robust algorithms resulting from these equilibria. This study is challenging as determining the performances of iterative distributed construction algorithms such as the generic algorithm in Section 1.1.2 is non trivial, even if nodes are altruistic, and follow the prescribed actions imposed by the algorithm. On the other hand, this line of study is of the utmost importance as, in the framework of large scale distributed computing, it is unreasonable to assume that no nodes will be tempted to deviate from the prescribed actions, for optimizing its own benefit, at the expense of the performances of the algorithms, and of the quality of the solutions.

References

1. Ittai Abraham, Danny Dolev, and Joseph Y. Halpern. Distributed Protocols for Leader Election: A Game-Theoretic Perspective. In *27th International Symposium on Distributed Computing (DISC)*, pages 61–75, 2013. doi:10.1007/978-3-642-41527-2_5.
2. Daron Acemoglu, Munther A Dahleh, Ilan Lobel, and Asuman Ozdaglar. Bayesian learning in social networks. *The Review of Economic Studies*, 78(4):1201–1236, 2011.
3. Yehuda Afek, Yehonatan Ginzberg, Shir Landau Feibish, and Moshe Sulamy. Distributed computing building blocks for rational agents. In *Symposium on Principles of Distributed Computing (PODC)*, pages 406–415. ACM, 2014.
4. Yehuda Afek, Shaked Rafaeli, and Moshe Sulamy. Cheating by Duplication: Equilibrium Requires Global Knowledge. Technical report, arXiv, 2017. arXiv:1711.04728.
5. Noga Alon, László Babai, and Alon Itai. A Fast and Simple Randomized Parallel Algorithm for the Maximal Independent Set Problem. *J. Algorithms*, 7(4):567–583, 1986. doi:10.1016/0196-6774(86)90019-2.
6. Chen Avin, Avi Cohen, Pierre Fraigniaud, Zvi Lotker, and David Peleg. Preferential Attachment as a Unique Equilibrium. In *The Web Conference (WWW)*, New-York, 2018. ACM.
7. Leonid Barenboim and Michael Elkin. *Distributed Graph Coloring: Fundamentals and Recent Developments*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2013.
8. A. Calvó-Armengol and J. de Martí Beltran. Information gathering in organizations: Equilibrium, welfare and optimal network structure. *Journal of the European Economic Association*, 7:116–161, 2009.
9. A. Calvó-Armengol, J. de Martí Beltran, and Prat A. Communication and influence. *Theoretical Economics*, 10:649–690, 2015.
10. Subir K Chakrabarti. Equilibrium in Behavior Strategies in Infinite Extensive Form Games with Imperfect Information. *Economic Theory*, 2(4):481–494, 1992.
11. Ky Fan. Fixed-Point and Minimax Theorems in Locally Convex Topological Linear Spaces. *PNAS*, 38(2):121–126, 1952.
12. Drew Fudenberg and David Levine. Subgame-Perfect Equilibria of Finite- and Infinite-Horizon Games. *Journal of Economic Theory*, 31(2):251–268, 1983.

13 Andrea Galeotti, Christian Ghiglino, and Francesco Squintani. Strategic information transmission networks. *J. Economic Theory*, 148(5):1751–1769, 2013.

14 Irving L. Glicksberg. A Further Generalization of the Kakutani Fixed Point Theorem with Application to Nash Equilibrium Points. *Proceedings of the AMS*, 3(1):170–174, 1952.

15 J. Hagenbach and F. Koessler. Strategic communication networks. *Review of Economic Studies*, 77(3):1072–1099, 2011.

16 Christopher Harris. Existence and Characterization of Perfect Equilibrium in Games of Perfect Information. *Econometrica: Journal of the Econometric Society*, 53(3):613–628, 1985.

17 Matthew O. Jackson, Brian W. Rogers, and Yves Zenou. Networks: An Economic Perspective. Technical report, arXiv, 2016.

18 Matthew O Jackson and Yves Zenou. Games on networks. In *Handbook of game theory with economic applications*, volume 4, pages 95–163. Elsevier, 2015.

19 Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. Radio Network Clustering from Scratch. In *12th European Symposium on Algorithms (ESA)*, LNCS 3221, pages 460–471. Springer, 2004.

20 Harold W. Kuhn. Extensive Games and the Problem of Information. In *Contributions to the Theory of Games*, volume II (Annals of Mathematics Studies, 28), pages 193–216. Princeton University Press, 1953.

21 Nathan Linial. Locality in Distributed Graph Algorithms. *SIAM J. Comput.*, 21(1):193–201, 1992.

22 Michael Luby. A Simple Parallel Algorithm for the Maximal Independent Set Problem. *SIAM J. Comput.*, 15(4):1036–1053, 1986. doi:10.1137/0215074.

23 Thomas Moscibroda and Roger Wattenhofer. Maximal independent sets in radio networks. In *24th Symposium on Principles of Distributed Computing (PODC)*, pages 148–157. ACM, 2005. doi:10.1145/1073814.1073842.

24 Moni Naor and Larry J. Stockmeyer. What Can be Computed Locally? *SIAM J. Comput.*, 24(6):1259–1277, 1995. doi:10.1137/S0097539793254571.

25 John F. Nash. Equilibrium Points in n-Person Games. *Proceedings of the National Academy of Sciences*, 36(1):48–49, 1950.

26 David Peleg. *Distributed Computing: A Locality-Sensitive Approach.* Discrete Mathematics and Applications. SIAM, Philadelphia, 2000.

27 Roy Radner. Collusive Behavior in non-Cooperative epsilon-Equilibria of Oligopolies with Long but Finite Lives. *Journal of Economic Theory*, 22:136–154, 1980.

28 Reinhard Selten. Spieltheoretische Behandlung eines Oligopolmodells mit Nachfragetraheit. *Z. Gesamte Staatwissenschaft*, 12:301–324, 1965.

29 Reinhard Selten. Reexamination of the Perfectness Concept for Equilibrium Points in Extensive Games. *International journal of game theory*, 4(1):25–55, 1975.

The Sparsest Additive Spanner via Multiple Weighted BFS Trees

Keren Censor-Hillel
Department of Computer Science, Technion, Haifa, Israel
ckeren@cs.technion.ac.il

Ami Paz
IRIF, CNRS and Paris Diderot University, Paris, France
amipaz@irif.fr

Noam Ravid
Department of Computer Science, Technion, Haifa, Israel
noamrvd@cs.technion.ac.il

Abstract

Spanners are fundamental graph structures that sparsify graphs at the cost of small stretch. In particular, in recent years, many sequential algorithms constructing additive all-pairs spanners were designed, providing very sparse small-stretch subgraphs. Remarkably, it was then shown that the known (+6)-spanner constructions are essentially the sparsest possible, that is, larger additive stretch cannot guarantee a sparser spanner, which brought the stretch-sparsity trade-off to its limit. Distributed constructions of spanners are also abundant. However, for additive spanners, while there were algorithms constructing (+2) and (+4)-all-pairs spanners, the sparsest case of (+6)-spanners remained elusive.

We remedy this by designing a new sequential algorithm for constructing a (+6)-spanner with the essentially-optimal sparsity of $\tilde{O}(n^{4/3})$ edges. We then show a distributed implementation of our algorithm, answering an open problem in [10].

A main ingredient in our distributed algorithm is an efficient construction of multiple weighted BFS trees. A weighted BFS tree is a BFS tree in a weighted graph, that consists of the lightest among all shortest paths from the root to each node. We present a distributed algorithm in the CONGEST model, that constructs multiple weighted BFS trees in $|S| + D - 1$ rounds, where S is the set of sources and D is the diameter of the network graph.

2012 ACM Subject Classification Theory of computation → Distributed computing models, Theory of computation → Sparsification and spanners, Theory of computation → Shortest paths

Keywords and phrases Distributed graph algorithms, congest model, weighted BFS trees, additive spanners

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.7

Related Version The full version of this paper is available on the arXiv [11], https://arxiv.org/abs/1811.01997.

Funding This project has received funding from the European Union's Horizon 2020 Research And Innovation Programe under grant agreement no. 755839, and also partially supported by ISF individual research grant 1696/14. Ami Paz was supported by the Fondation Sciences Mathématiques de Paris (FSMP).

Acknowledgements We thank Shiri Chechik and Pierre Fraigniaud for discussions regarding (+6)-spanners.

© Keren Censor-Hillel, Ami Paz, and Noam Ravid;
licensed under Creative Commons License CC-BY
22nd International Conference on Principles of Distributed Systems (OPODIS 2018).
Editors: Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira; 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

A spanner of a graph G is a spanning subgraph H of G that approximately preserves distances. Spanners find many applications in distributed computing [12, 9, 42, 43, 45], and thus their distributed construction is the center of many research papers. We focus on spanners that approximately preserve distances between all pairs of nodes, and where the stretch is only by an additive factor (*purely-additive all-pairs* spanners).

Out of the abundant research on distributed constructions of spanners, only two papers discuss the construction of *purely additive* spanners in the CONGEST model: the construction of (+2)-spanners is discussed in [35], and the construction of (+4)-spanners and (+8)-spanners in [10], along with other types of additive spanners and lower bounds. However, the distributed construction of (+6)-spanners remained elusive, stated explicitly as an open question in [10]. This is especially important since additive factors greater then 6 cannot yield essentially sparser spanners [2].

In this paper, we give a distributed algorithm for constructing a (+6)-spanner, with an optimal number of edges up to sub-polynomial factors; our spanner is even sparser than the (+8)-spanner presented in [10]. Several sequential algorithms building (+6)-spanners were presented, but none of them seems to be appropriate for a distributed setting. Thus, to achieve our result we also present a new, simple sequential algorithm for constructing (+6)-spanners, a result that could be of independent interest.

As a key ingredient, we provide a distributed construction of *multiple weighted BFS trees*. Constructing a breadth-first search (BFS) tree is a central task in many computational settings. In the classic synchronous distributed setting, constructing a BFS tree from a given source is straightforward. Due to its importance, this task has received much attention in additional distributed settings, such as the asynchronous setting (see, e.g., [40] and references therein). Moreover, at the heart of many distributed applications lies a graph structure that represents the edges of *multiple* BFS trees [30, 34], which are rooted at the nodes of a given subset $S \subseteq V$, where $G = (V, E)$ is the underlying communication graph. Such a structure is used in distance computation and estimation [30, 29, 34], routing table construction [34], spanner construction [10, 35, 34], and more.

When the bandwidth is limited, constructing multiple BFS trees efficiently is a non-trivial task. Indeed, distributed constructions of multiple BFS trees in the CONGEST model [40], where in each round of communication every node can send $O(\log n)$-bit messages to each of its neighbors, have been given in [30, 34], who showed that it is possible to build BFS trees from a set of sources S in $O(|S| + D)$ rounds, where D is the diameter of the graph G. It is easy to show that this is asymptotically tight.

In some cases, different edges of the graph may have different attributes, which can be represented using edge weights. The existence of edge weights has been extensively studied in various tasks, such as finding or approximating lightest paths [20, 37, 27, 21, 34, 31, 4, 25], finding a minimum spanning tree (MST) in the graph [5, 23, 13], finding a maximum matching [36, 13], and more. However, as far as we are aware, no study addresses the problem of constructing multiple weighted BFS (WBFS) trees, where the goal is not to find the lightest paths from the sources to the nodes, but rather the *lightest shortest paths*. That is, the path in a WBFS tree from the source s to a node v is the lightest among all shortest paths from s to v in G.

Thus, we provide an algorithm that constructs multiple WBFS trees from a set of source nodes S in the CONGEST model. Our algorithm completes in $|S| + D - 1$ rounds, which implies that no overhead is needed for incorporating the existence of weights.

1.1 Our contribution

At a high level, our approach for building multiple WBFS trees is to generalize the algorithm of Lenzen et al. [34] in order to handle weights. In [34], the messages are pairs consisting of a source node and a distance, which are prioritized by the distance traversed so far. When incorporating weights into this framework it makes sense to use triplets instead of pairs, where each triplet also contains the weight of the respective path. However, it may be that a node v needs to send multiple messages that correspond to the same source and the same distance but contain different weights, since congestion over edges may cause the respective messages to arrive at v in different rounds and, in the worst case, in a decreasing order of weights. The challenge in generalizing this framework therefore lies in guaranteeing that despite the need to consider weights, we can carefully choose a total order to prioritize triplets, such that not too many messages need to be sent, allowing us to handle congestion. Our construction and its proof appear in Section 3, giving the following.

▶ **Theorem 1.** *Given a weighted graph $G = (V, E, w)$ and a set of nodes $S \subseteq V$, there exists an algorithm for the* CONGEST *model that constructs a WBFS tree rooted at s, for every $s \in S$, in $|S| + D - 1$ rounds.*

The importance of our multiple WBFS trees construction lies in our ability to use it for pinning down the question of constructing (+6)-spanners in the CONGEST model. The construction of additive spanners in the CONGEST model was studied beforehand [35, 10], but the +6 case remained unresolved, for reasons we describe below. Naturally, the quality of a spanner is measured by its sparsity, which is the motivation for allowing some stretch in the distances to begin with, and different spanners present different tradeoffs between stretch and sparsity. The properties of our (+6)-spanner construction algorithm are summarized in the following theorem.[1]

▶ **Theorem 2.** *There exists an algorithm for the* CONGEST *model that constructs a (+6)-spanner with $O\left(n^{4/3}\log^{4/3} n\right)$ edges in $O\left(\frac{n^{2/3}}{\log^{1/3} n} + D\right)$ rounds and sucseeds w.h.p.*

Previous distributed algorithms for spanners similar to ours, i.e., purely additive all-pairs spanners, construct a (+2)-spanner with $\tilde{O}(n^{3/2})$ edges in $\tilde{O}(n^{1/2} + D)$ rounds [35], a (+4)-spanner with $\tilde{O}(n^{7/5})$ edges in $\tilde{O}(n^{3/5} + D)$ rounds [10], and a (+8)-spanner with $\tilde{O}(n^{15/11})$ edges in $\tilde{O}(n^{7/11} + D)$ rounds [10]. Hence, our algorithm is currently the best non-trivial spanner construction algorithm in terms of density, sparser even than the previous (+8)-spanner. The option of getting even sparser spanners by allowing more stretch was essentially ruled out [2], while the question of improving the running time remains open for all stretch parameters.

1.2 Other spanner construction algorithms

Previous distributed spanner construction algorithms all build upon known sequential algorithms, and present a distributed implementations of them, or of a slight variant of them [35, 10]. For example, many sequential algorithms start in a clustering phase, where stars around high-degree nodes are added to the spanner one by one. Implementing this directly in the distributed setting will take too long; instead, it is shown that choosing cluster centers at random yields almost as good results, and can be implemented in a constant

[1] We use w.h.p. to indicate a probability that is at least $1 - 1/n^c$ for some constant $c \geq 1$ of choice.

time. Similar methods are used for implementing other parts of the construction. However, the approach of finding a distributed implementation for a sequential algorithm fails for all known (+6)-spanner algorithms, as described next. Thus, we introduce a new sequential algorithm for the problem, and then present its distributed implementation.

There are three known approaches for the design of sequential (+6)-spanner algorithms. The first, presented by Baswana et al. [6], is based on measuring the quality of paths in terms of *cost* and *value*, and adding to the spanner only paths which are "affordable". This approach was later extended by Kavitha [32] to other families of additive spanners. The second approach, presented by Woodruff [46], uses a subroutine that finds almost-shortest paths between pairs of nodes, and obtains a faster algorithm at the expense of a slightly worst sparsity guarantee. The third approach, presented by Knudsen [33], is based on repeatedly going over pairs of nodes, and adding a shortest path between a pair of nodes to the spanner if their current distance in the spanner is too large.

Unfortunately, direct implementation in the CONGEST model of the known sequential algorithms is highly inefficient. We are not aware of fast distributed algorithms that allow the computation of the cost and value of paths needed for the algorithm of [6]. Similarly, for [46], the almost-shortest paths subroutine seems too costly for the CONGEST model. The algorithm of [33] needs repeated updates of the distances in the spanner between pairs of nodes after every addition of a path to it, which is a sequential process in essence, and thus we do not find it suitable for an efficient distributed implementation.

A different approach for the distributed construction of (+6)-spanners could be to adapt a distributed algorithm with different stretch guarantees to construct a (+6)-spanner. This approach does not seem to work: the distributed algorithms for constructing (+2)-spanners [35] and (+4)-spanner [10] are both very much tailored for achieving the desired stretch, and it is not clear how to change them in order to construct sparser spanners with higher stretch. The (+8)-spanner construction algorithm [10] starts with clustering, and then constructs a (+4)-*pairwise spanner* between the cluster centers. Replacing the (+4)-pairwise spanner by a (+2)-pairwise spanner will indeed yield a (+6)-all-pairs spanner, as desired. However, even using the sparsest (+2)-pairwise spanners [10, 1], the resulting (+6)-spanner may have $\tilde{O}(n^{5/3})$ edges, denser than our new (+6)-spanner and than the known (+8)-spanner [10].

Thus, we start by presenting a new sequential algorithm for the construction of (+6)-spanners, an algorithm that is more suitable for a distributed implementation, and then discuss its distributed implementation. Our construction starts with a clustering phase, and then adds paths that minimize the number of additional edges that need to be added to the spanner. To implement our construction in the CONGEST model, we assign weights to the edges and use our WBFS algorithm to find shortest paths with as few edges as possible that are not yet in the spanner. Note that although the graph and the spanner we construct for it are both unweighted, the ability of our multiple WBFS algorithm to handle weights is crucial for our solution.

A (+6)-spanner must contain $n^{4/3}/2^{O(\sqrt{\log n})}$ edges [2]. The best sequential algorithms [33, 6] construct a spanner with $O(n^{4/3})$ edges. Our distributed algorithm constructs a spanner with $O(n^{4/3}\log^{4/3} n)$ edges, which is slightly denser than optimal but still sparser than the $O(n^{4/3}\log^3 n)$ edges in the fast sequential construction of [46].

1.3 Related work

Algorithms for the CONGEST model that construct multiple (unweighted) BFS trees, rooted at a set of sources S, were suggested in [34] and [30], running in $O(|S| + D)$ rounds. Both algorithms start the construction of all the BFS trees simultaneously, and proceed by

transferring messages containing the source of a BFS tree and the distance the message has traversed so far. The algorithms differ in how they order message deliveries when several messages need to be sent over an edge at the same round. We base our multiple WBFS construction on the [34] algorithm, in which messages sent by a node are prioritized by the distance they traversed so far, with a preference to messages that traversed smaller distance. The [30] algorithm, which we cannot use for our construction [28], prioritizes messages by the identity of the root, and transmits a message only in one direction of each edge in each round.

Spanners were first introduced in 1989 by [41, 42], and since then have been a topic for wide research due to their abundant applications. Prime examples for the need for sparse spanners can be found in synchronizing distributed networks [42], information dissemination [9], compact routing schemes [12, 43, 45], and more.

Distributed constructions of various spanners have been widely studied [35, 34, 44, 10, 6, 7, 14, 15, 16, 18, 19, 22, 17, 39, 24, 38, 26, 8]. Lower bounds were given in [44, 10, 3]. However, obtaining an efficient and sparse (+6)-all-pairs spanner has remained an open question [10].

Several lower bounds for the time complexity of spanner construction in the CONGEST model where presented in [10], but these are applicable only to pairwise spanners with a bounded number of pairs, and not to all-pairs spanners. A lower bound from [44] states that the construction of a spanner with $\tilde{O}(n^{4/3})$ edges, such as the one we build, must take $\tilde{\Omega}(n^{3/8})$ rounds. This lower bound does not take into account the bandwidth restrictions at all (it is proven for the LOCAL model), and so we believe that a higher lower bound for the CONGEST model should apply, but this is left as an intriguing open question.

2 Preliminaries

All graphs in this work are simple, connected and undirected. A graph can be unweighted, $G = (V, E)$, or weighted $G = (V, E, w)$ with $w : E \to \{0, \ldots, W\}$, in which case we assume $W \in \text{poly}(n)$. Given a path ρ in a weighted graph G, we use $|\rho|$ to denote the *length* of ρ, which is the number of edges in it, and $w(\rho)$ to denote the *weight* of the path, which is the sum of its edge-weights. The *distance* between two nodes u, v in a graph G, denoted $\delta_G(u, v)$, is the minimum length of a path in G connecting u and v. The *diameter* of a graph (weighted or unweighted) is $D = \max_{u,v\in V} \{\delta_G(u, v)\}$.

We consider the CONGEST model of computation[40], where the nodes of a graph communicate synchronously by exchanging $O(\log n)$-bit messages along the edges. The goal is to distributively solve a problem while minimizing the number of communication rounds.

WBFS trees: We are interested in a weighted BFS tree, which consists of all *lightest shortest paths* from the root, formally defined as follows.

▶ **Definition 3.** Given a connected, weighted graph $G = (V, E, w)$ and a node $s \in V$, a *weighted BFS tree (WBFS)* for G rooted at s is a spanning tree T_s of G satisfying the following properties:

(i) For each $v \in V$, the path from s to v in T is a shortest path in G between s and v.

(ii) For each $v \in V$, no shortest path from s to v in G is lighter than the path from s to v in T.

We emphasize that this is different than requiring a subgraph containing all *lightest paths* from the root. One may wonder if a WBFS tree always exists, but this is easily evident by the following refinement of a (sequential) BFS search, returning a WBFS tree: go over the

nodes in an order of non-decreasing distances from the source s, starting with $w(s) = 0$; each node v chooses as a parent a neighbor u that was already processed and minimizes $w(v) = w(u)+w(u, v)$, and adds the edge $\{u, v\}$ to the tree. Each node has a single parent, so this is indeed a tree; the node ordering guarantees that this is indeed a BFS tree, assuring (i); and the parent choice guarantees the paths are lightest among the shortest, assuring (ii).

Spanners: Given a graph $G = (V, E)$, a subgraph $H = (V, E')$ of G is called an (α, β)-*spanner* if for every $u, v \in V$ it holds that $\delta_H(u, v) \leq \alpha\delta_G(u, v) + \beta$. The parameters α and β are called the *stretch parameters*.

When $\alpha = 1$, such a spanner is called a *purely additive spanner*. In this paper we focus on purely additive (+6)-spanners, i.e., $\alpha = 1$ and $\beta = 6$.

For completeness, we mention that when $\beta = 0$, such a spanner is called a *multiplicative spanner*. In addition, while sometimes the stretch parameters need to be guaranteed only for some subset of all the pairs of nodes of the graph (such as in *pairwise spanners*), we emphasize that our construction provides the promise of a +6 stretch for *all* pairs.

3 Multiple Weighted BFS Trees

In the CONGEST model, the problem of finding a WBFS tree requires each node to know its parent in the WBFS tree, and the unweighted and weighted distances to the source within the tree. This allows the node to send messages to the source node through the lightest among all shortest paths. When there are multiple sources, each node should know the parent leading to each of the sources in S.

We define data structures for representing multiple WBFS trees. Given a node $v \in V$, the *S-proximity-list* (or *proximity list* for short) of v, noted $\texttt{PL}^*_v$, is an ascending lexicographically ordered list of triples $(d(s, v), s, w(s, v))$, where $d(s, v)$ and $w(s, v)$ are the length and weight of the path from s to v in T_s. Two different triples are ordered such that $(d(s, v), s, w(s, v)) < (d(t, v), s, w(t, v))$ if $d(s, v) < d(t, v)$, or $d(s, v) = d(t, v)$ and $s < t$, where s and t may be compared by any predefined order on the node identifiers. Note that T_s contains a single path from s to v, so both $d(s, v) = d(t, v)$ and $s = t$ cannot happen simultaneously without having $w(s, v) = w(t, v)$.

The *S-path-map* (or *path-map* for short) of v is a mapping from each source $s \in S$ to the parent of v in T_s, noted by $\texttt{PM}^*_v$. The list $\texttt{PM}^*_v$ is sorted with respect to the order of $\texttt{PL}^*_v$, such that the first records of $\texttt{PM}^*_v$ belong to sources closest to v.

Algorithm 1, which constructs multiple WBFS trees from a set S in the CONGEST model, is based on carefully extending the distributed Bellman-Ford-based algorithm of Lenzen et al. [34]. The heart of the algorithm is a loop (Line 7), and each iteration of it takes a single round in the CONGEST model. We show that $|S| + D - 1$ iterations of the loop suffice in order to construct the desired WBSF trees.

The algorithm builds the WBFS trees by gradually updating the proximity list and the path map of each node. Each round is composed of two phases: updating the neighbors about changes in the proximity list, and receiving updates from other nodes. The path map is only used by the current node, and therefore changes to it are not sent.

Ideally, each node would update its neighbors regarding all the changes made to its proximity list. However, due to bandwidth restrictions, a node cannot send the entire list in each round. Therefore, at each round each node sends to all of its neighbors the lexicographically smallest triplet in its proximity list that it has not yet sent, while maintaining a record noting which triplets have been sent and which are waiting. Each triplet is only sent once, though a node may send multiple triplets regarding a single source.

Algorithm 1: Weighted distributed Bellman-Ford algorithm for node v

```
1  L_v ← ()
2  for s ∈ S do
3  |  PM_v(s) ← ⊥
4  if v ∈ S then
5  |  PL_v ← ((0, v, 0))
6  |  sent_v(0, v, 0) ← FALSE          /* A variable marking sent triplets */
7  for |S| + D − 1 rounds do
8  |  if ∃(d_s, s, w_s) ∈ PL_v  such that sent_v(d_s, s, w_s) = FALSE then
9  |  |  (d_s, s, w_s) ← min {(d_t, t, w_t) ∈ PL_v  such that sent_v(d_t, t, w_t) = FALSE}
10 |  |  send (d_s, s, w_s) to all neighbors
11 |  |  sent_v(d_s, s, w_s) ← TRUE
12 |  for received (d_s, s, w_s) from u ∈ V do
13 |  |  d_s ← d_s + 1
14 |  |  w_s ← w_s + w(u, v)
15 |  |  if ∄(d'_s, s, w'_s) ∈ PL_v  such that (d'_s < d_s or (d'_s = d_s and w'_s < w_s)) then
16 |  |  |  PL_v ← PL_v \ {(·, s, ·)}
17 |  |  |  PL_v ← PL_v ∪ {(d_s, s, w_s)}
18 |  |  |  PM_v(s) ← u
19 |  |  |  sent_v(d_s, s, w_s) ← FALSE
```

A node uses the messages received in the current round in order to update its proximity list and path map for the next round. A triplet (d_s, s, w_s) received by a node v from a neighbor u represents the length d_s and weight w_s of some path ρ from s to u in the graph. The node v then considers the extended path $\rho' = \rho \circ v$ from s to v, compares it to its currently known best path from s to v, and updates the proximity list and path map in case a shorter path has been found, or a lighter path with the same length.

To prove correctness, we generalize the proof of [34] to handle weights, and show that our algorithm solves the *weighted* (S, d, k)*-detection* problem: each node should learn which are the sources from S closest to it, but at most k of them and only up to distance d. This is formally defined as follows.

▶ **Definition 4.** Given a weighted graph $G = (V, E, w)$, a subset $S \subseteq V$ of source nodes, and a node $v \in V$, let $\mathtt{PL}^*_v$ denote the S-proximity-list and let $\mathtt{PM}^*_v$ denote the path map of the node v. The *weighted* (S, d, k)*-detection problem* requires that each node $v \in V$ learns the first $\min\{k, \lambda^d_v\}$ entries of $\mathtt{PL}^*_v$ and $\mathtt{PM}^*_v$, where λ^d_v is the number of sources $s \in S$ such that $d(s, v) \leq d$.

Given a node v, $\mathtt{PL}_v$ is a variable in Algorithm 1 holding the proximity list of v, and we denote by $\mathtt{PL}^{(r)}_v$ the state of the list $\mathtt{PL}_v$ at the beginning of round r of the algorithm, and by $\mathtt{PL}^{(\infty)}_v$ the value of $\mathtt{PL}_v$ at the end of the algorithm. Recall that $\mathtt{PL}^*_v$ is the true proximity list, so our goal is proving $\mathtt{PL}^{(\infty)}_v = \mathtt{PL}^*_v$, i.e., proving that the algorithm obtains the correct values of the proximity list.

We use similar notations for the path map $\mathtt{PM}^*_v$. Since the records of $\mathtt{PM}_v$ are updated under the same conditions as the records of $\mathtt{PL}_v$, the correctness of $\mathtt{PM}_v$ at the end of the algorithm with respect to $\mathtt{PM}^*_v$ immediately follows, and we omit the details.

We start by showing that if there was no bound on the number of rounds, then the values of $\texttt{PL}_v$ would have eventually converged to the true values of $\texttt{PL}^*_v$. The proof of Lemma 5 can be found at the full version of this paper [11].

▶ **Lemma 5.** *Given a graph $G = (V, E, w)$ and a set $S \subseteq V$, if we let the for loop in Line 7 of Algorithm 1 to run forever, then there exists a round $r_0 \in \mathbb{N}$ such that no node $v \in V$ sends messages or modifies $\texttt{PL}_v$ after round r_0. Moreover, $\texttt{PL}^{(r_0)}_v = \texttt{PL}^*_v$, i.e., for every $(d_s, s, w_s) \in \texttt{PL}^{(r_0)}_v$, it holds that $d_s = d(v, s)$ and $w_s = \min\{w(\rho) \mid \rho$ connects v with s, and $|\rho| = d_s\}$.*

Lemma 5 shows that without the limit on the number of rounds, the algorithm would compute the right values; however, it does not bound the number of rounds needed for this to occur. Next, we show that $|S| + D - 1$ rounds suffice. We cannot apply the claims of [34] directly, since the existence of weights restricts the number of viable solutions even further, causing more updates to the proximity list and an increase in the number of messages sent. However, we do use a similar technique: we bound the number of rounds in which the k smallest entries of $\texttt{PL}_v$ can change.

For an entry $(d_s, s, w_s) \in \texttt{PL}^{(r)}_v$, let $\ell^{(r)}_v(d_s, s, w_s)$ denote the index of the entry in the lexicographically ordered list $\texttt{PL}^{(r)}_v$ at the beginning of round r. For completeness, we define $\ell^{(r)}_v(d_s, s, w_s) = -\infty$ if (d_s, s, w_s) did not appear in $\texttt{PL}_v$ at the beginning of round r, and $\texttt{PL}_v = \infty$ if the triplet was removed from $\texttt{PL}_v$ before the beginning of this round. Note that a removed triplet is never returned to the list, since the lexicographical order is transitive.

▶ **Lemma 6.** *For a triplet (d_s, s, w_s), the following holds:*
(i) *$\ell^{(r)}_v(d_s, s, w_s)$ is non-decreasing with r.*
(ii) *If the triplet (d_s, s, w_s) is sent from a node u to a node v at round r, resulting in the addition of a new triplet (d'_s, s, w'_s) to $\texttt{PL}_v$ at the end of round r, where $d'_s = d_s + 1$ and $w'_s = w_s + w(u, v)$, then $\ell^{(r)}_u(d_s, s, w_s) \le \ell^{(r+1)}_v(d'_s, s, w'_s)$.*

Part (i) follows from the fact that the number of triplets below (d_s, s, w_s) cannot decrease. To prove part (ii), we show that all the triplets below (d_s, s, w_s) in $\texttt{PL}_u$ are sent from u to v and added to $\texttt{PL}_v$ before (d_s, s, w_s) is sent and added.

Proof. Part (i) is a consequence of the method used by our algorithm for managing the list $\texttt{PL}_v$. According to our algorithm, triplets are not removed from $\texttt{PL}_v$ when they are sent. The only case in which a triplet (d_t, t, w_t) is removed from $\texttt{PL}_v$ is when a lexicographically smaller triplet (d'_t, t, w'_t) is added to the list instead. When this happens in round r, it holds that $\ell^{(r)}_v(d_t, t, w_t) \ge \ell^{(r+1)}_v(d'_t, t, w'_t)$, since the new triplet is lexicographically smaller. Hence, for every other triplet $(d_s, s, w_s) \in \texttt{PL}_v$, the number of lexicographically smaller triplets in $\texttt{PL}_v$ cannot decrease throughout the algorithm.

We now turn to prove part (ii) of the lemma. By the fact that the triplet (d_s, s, w_s) is sent by the node u in round r, we conclude that the $\ell^{(r)}_u(d_s, s, w_s) - 1$ triplets preceding it in the list $\texttt{PL}^{(r)}_u$ have already been sent by u in earlier rounds, and arrived at the node v. For each such triplet (d_t, t, w_t), either $d_t \le d_s$, or $t < s$ and $d_t = d_s$. Therefore, when added to $\texttt{PL}_v$ as $(d_t + 1, t, w_t + w(u, v))$ it is lexicographically smaller than (d'_s, s, w'_s). At round r, either $(d_t + 1, t, w_t + w(u, v))$ is in $\texttt{PL}^{(r)}_v$ or it was replaced by a lexicographically smaller triplet containing t. Thus, there are at least $\ell^{(r)}_u(d_s, s, w_s) - 1$ triplets smaller than (d'_s, s, w'_s) in $\texttt{PL}^{(r+1)}_v$, and hence $\ell^{(r)}_u(d_s, s, w_s) \le \ell^{(r+1)}_v(d'_s, s, w'_s)$. ◀

Lemma 6 implies that as the algorithm progresses, messages at higher indexes of the proximity list are sent and updated. This can be used to obtain an upper bound on the round in which a triplet at a certain index of the proximity list can be sent or received, as formalized by the next lemma.

▶ **Lemma 7.** *In round $r \in \mathbb{N}$ of Algorithm 1, a node $v \in V$ can:*

(i) *send a message (d_s, s, w_s) only if*

$$d_s + \ell_v^{(r)}(d_s, s, w_s) \geq r$$

(ii) *add to $\mathtt{PL}_v$ a triplet (d_s, s, w_s) only if*

$$d_s + \ell_v^{(r+1)}(d_s, s, w_s) > r$$

Part (i), when put in words, is rather intuitive: while a triplet might need to wait before being sent, the waiting time is bounded from above by the distance the triplet has traversed from its source, plus the number of triplets that were to be sent before it. Part (ii) is complementary to part (i): the time before a triplet is added, is, once more, bounded by the distance it traversed plus the number of lexicographically smaller triplets.

Proof. We start by showing that, for a given round r, if Lemma 7(i) holds for all nodes then Lemma 7(ii) holds as well. Consider a triplet (d'_s, s, w'_s) that is added to $\mathtt{PL}_v$ as a result of a message (d_s, s, w_s) sent from u to v in round r, where $d'_s = d_s + 1$ and $w'_s = w_s + w(u, v)$. Lemma 7(i) implies that $d_s + \ell_u^{(r)}(d_s, s, w_s) \geq r$, and by Lemma 6(ii) we have that $\ell_v^{(r+1)}(d'_s, s, w'_s) \geq \ell_u^{(r)}(d_s, s, w_s)$. As $d'_s > d_s$, we conclude

$$d'_s + \ell_v^{(r+1)}(d'_s, s, w'_s) > d_s + \ell_u^{(r)}(d_s, s, w_s) \geq r,$$

which implies Lemma 7(ii).

Next, we prove by induction that both parts of the lemma hold. In round 1, Lemma 7(i) holds trivially, since by definition $\ell_v^{(1)}(d_s, s, w_s) \geq 1$. Assume that Lemma 7 holds at round $r - 1$; we show the lemma holds at round r. Since Lemma 7(i) implies Lemma 7(ii), it is sufficient to show that every message (d_s, s, w_s) sent by some node $v \in V$ in round r satisfies $d_s + \ell_v^{(r)}(d_s, s, w_s) \geq r$.

Observe that if (d_s, s, w_s) is sent by a node v in round r, then the triplet must have been added to $\mathtt{PL}_v$ in some round $r' \leq r - 1$. If $r' = r - 1$, according to the induction hypothesis, Lemma 7(ii) holds and $d_s + \ell_v^{(r)}(d_s, s, w_s) > r - 1$, implying $d_s + \ell_v^{(r)}(d_s, s, w_s) \geq r$, since all the terms are integers.

Otherwise $r' < r - 1$. In this case, in round $r - 1$ the triplet (d_s, s, w_s) appeared in $\mathtt{PL}_v$ and was not yet sent. Since (d_s, s, w_s) is sent in round r, a different triplet (d_t, t, w_t) with $t \neq s$ must have been sent in round $r - 1$, implying:

$$d_s + \ell_v^{(r-1)}(d_s, s, w_s) > d_t + \ell_v^{(r-1)}(d_t, t, w_t).$$

By Lemma 6(i), we have that $\ell_v^{(r)}(d_s, s, w_s) \geq \ell_v^{(r-1)}(d_s, s, w_s)$, and combined with the induction hypothesis for Lemma 7(i) in round $r - 1$ we conclude:

$$d_s + \ell_v^{(r)}(d_s, s, w_s) \geq d_s + \ell_v^{(r-1)}(d_s, s, w_s) > d_t + \ell_v^{(r-1)}(d_t, t, w_t) \geq r - 1.$$

This gives that $d_s + \ell_v^{(r)}(d_s, s, w_s) \geq r$, since all the terms are integers. ◀

Lemma 5 implies that eventually, the lists $\mathtt{PL}_v$ converge to contain the correct values, and Lemma 7 restricts the number of rounds in which specific list entries may change. From this, we conclude that the algorithm solves the weighted (S, d, k)-detection problem.

▶ **Lemma 8.** *Given an instance of the weighted (S, d, k)-detection problem, for every $v \in V$ and round r of an execution of Algorithm 1 with*

$$r \geq \min\{d, D\} + \min\{k, |S|\},$$

the truncation of $\mathtt{PL}_v^{(r)}$ *to the first* $\min\{k, \lambda_v^d\}$ *entries, where* λ_v^d *is the number sources* $s \in S$ *such that* $d(s,v) \le d$*, solves weighted* (S,d,k)*-detection problem.*

This lemma says that the truncated list is correct at the beginning of the relevant round. To prove it, we use Lemma 7(ii) to show that the values in the truncated list cannot change at round r or later, and Lemma 5 to deduce they are correct.

Proof. Assume w.l.o.g that $d \le D$, as D bounds the distance to any source, and $k \le |S|$, as otherwise v needs to learn about all sources.

By Lemma 5, there is a round r_0 when all entries of $\mathtt{PL}_v^{(r_0)}$ are correct, and let (d_s, s, w_s) be a triplet in one of the first $\min\{k, \lambda_v^d\}$ entries of $\mathtt{PL}_v^{(r_0)}$. Since (d_s, s, w_s) is one of the first λ_v^d entries and $\mathtt{PL}_v^{(r_0)} = \mathtt{PL}_v^*$, we have $d_s \le d$.

Let r be the round when (d_s, s, w_s) is inserted to the list $\mathtt{PL}_v$. By Lemma 7(ii), $r < d_s + \ell_v^{(r+1)}(d_s, s, w_s)$. By Lemma 6(i), when the triplet is inserted to the list, it is already placed in one of the first $\min\{k, \lambda_v^d\}$ entries, i.e., $\ell_v^{(r+1)}(d_s, s, w_s) \le \min\{k, \lambda_v^d\} \le k$. Hence,

$$r < d_s + \ell_v^{(r+1)}(d_s, s, w_s) \le d + k.$$

Since this claim holds for any of the first $\min\{k, \lambda_v^d\}$ entries, these were all correct at the beginning of round $d+k$, and in all the succeeding rounds. ◀

The construction of multiple WBFS trees is an instance of the $(S, D, |S|)$-detection problem. Lemma 8 shows that after $|S| + D - 1$ rounds of Algorithm 1 on such an instance, all the entries of the list $\mathtt{PL}_v^{(|S|+D)}$ are correct, yielding the main result of this section.

Theorem 1 (restated). *Given a weighted graph* $G = (V, E, w)$ *and a set of nodes* $S \subseteq V$*, there exists an algorithm for the* CONGEST *model that constructs a WBFS tree rooted at* s*, for every* $s \in S$*, in* $|S| + D - 1$ *rounds.*

4 A (+6)-Spanner Construction

In this section we discuss the distributed construction of (+6)-spanners. First, we present a template for constructing a (+6)-spanner and analyze the stretch and sparsity of the constructed spanner. Then, we provide an implementation of our template in the CONGEST model and analyze its running time.

A *cluster* C_i around a *cluster center* $c_i \in V$ is a subset of the set of neighbors of c_i in G. A node belonging to a cluster is *clustered*, while the other nodes are *unclustered*.

Our algorithm starts by randomly choosing cluster centers, and adding edges between them to their neighbors, where each neighbor arbitrarily chooses a single center to connect to. Then, additional edges are added, to connect each unclustered node to all its neighbors. Next, shortest paths between clusters are added to the spanner. In order to find these shortest paths in the CONGEST model, we use the WBFS construction algorithm to build WBFS trees from random sources. At the heart of our algorithm stands the path-hitting framework of Woodruff [46]: a shortest path in the graph which has many edges between clustered nodes, must go through many clusters. This fact is used in order to show that a path with many missing edges (edges not in H) is more likely to have an adjacent source of a WBFS tree, and thus it is well approximated by a path within the spanner.

Woodruff's algorithm starts with a similar clustering step. However, in order to add paths between clusters, it uses an involved subroutine that finds light almost-shortest paths between pairs of nodes. This subroutine seems too global to be implemented efficiently in a

distributed setting, so in our construction it is replaced by only considering lightest shortest paths, which we do using the WBFS trees defined earlier.

Our algorithm constructs a (+6)-spanner with $O(n^{4/3}\log^{4/3} n)$ edges in $\tilde{O}(n^{2/3}+D)$ rounds, as stated next.

Theorem 2 (restated). *There exists an algorithm for the* CONGEST *model that constructs a* $(+6)$*-spanner with* $O\left(n^{4/3}\log^{4/3} n\right)$ *edges in* $O\left(\frac{n^{2/3}}{\log^{1/3} n}+D\right)$ *rounds and sucseeds w.h.p.*

Lemmas 9 and 10 analyze the size and stretch of Algorithm `6AP` given below. The number of rounds of its distributed implementation is analyzed in Lemma 11, giving Theorem 2. We use $c>2$ to denote a constant that can be chosen according to the desired exponent of $1/n$ in the failure probability.

Algorithm 6AP

Input: a graph $G=(V,E)$, a constant $c>2$;
Output: a subgraph H of G;
Initialization: $n \leftarrow |V|$; $H \leftarrow (V,\emptyset)$; $k \leftarrow 1$

Clustering. Pick each node as a *cluster center* w.p. $\frac{c}{n^{1/3}\log^{1/3} n}$, and denote the set of selected nodes by $\mathcal{C}=\{c_1,c_2,\ldots\}$. For each c_i, initialize a cluster $C_i \leftarrow \emptyset$.

For each node $v\in V$, choose a neighbor c_i of v which is a cluster center, if such a neighbor exists, add the edge (v,c_i) to H, and add v to C_i. If none of the neighbors of v is a cluster center, add to H all the edges adjacent to v. Let $H_0 \leftarrow H$.

Path Buying.
While $k \le \frac{8cn^{2/3}}{\log^{1/3} n}$ do:
1. $S_k \leftarrow \emptyset$
2. Add each cluster center $c_i \in \mathcal{C}$ to S_k w.p. $\frac{8c^2\log n}{k}$, independently of the other centers
3. For each pair $(c_i,c_j)\in \mathcal{C}\times S_k$:
   - **a.** $A \leftarrow \emptyset$ /* A is a set of paths */
   - **b.** For each $v \in C_j$:
     - **i.** Among all the shortest paths from c_i to v, let P_v be a path with minimum $|P_v \setminus H_0|$
     - **ii.** If $|P_v\setminus H_0|<2k$, add P_v to A
   - **c.** If $A\neq\emptyset$, add to H one of the shortest among the paths of A
4. $k \leftarrow 2k$

▶ **Lemma 9.** *Algorithm* `6AP` *outputs a subgraph* H *of* G *with* $O(n^{4/3}\log^{4/3} n)$ *edges, with probability at least* $1-O(n^{-c+1})$.

Proof. The algorithm starts with $H=(V,\emptyset)$ and only adds edges from G, so H is indeed a subgraph of G over the same node set.

In the first part of the clustering phase, each node adds to H at most one edge, connecting it to a single cluster center, for a total of $O(n)$ edges. Then, the probability that a node of degree at least $n^{1/3}\log^{4/3} n$ is left unclustered is at most $\left(1-\frac{c}{n^{1/3}\log^{1/3} n}\right)^{n^{1/3}\log^{4/3} n}$, which is $O(n^{-c})$. A union bound implies that all nodes of degree at least $n^{1/3}\log^{4/3} n$ are clustered w.p. $1-O(n^{-c+1})$, and thus the total number of edges added to H by unclustered nodes in the second part of the clustering phase is $O(n^{4/3}\log^{4/3} n)$, w.p. $1-O(n^{-c+1})$.

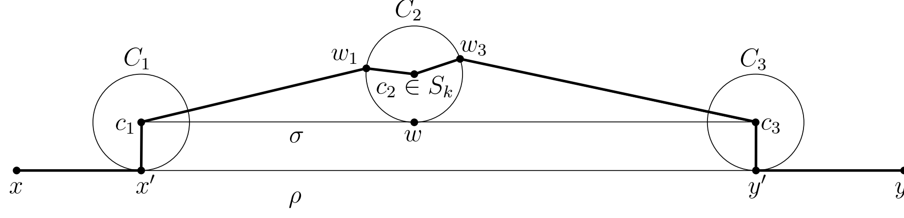


Figure 1 Illustration of the proof of Lemma 10

We start the analysis of the path buying phase by bounding the size of $\mathcal{C}$. A node $v \in V$ is added to $\mathcal{C}$ w.p. $\frac{c}{n^{1/3}\log^{1/3} n}$, so $\mathbb{E}[|\mathcal{C}|] = \frac{cn^{2/3}}{\log^{1/3} n}$. A Chernoff bound implies that

$$\Pr\left[|\mathcal{C}| > \frac{4cn^{2/3}}{\log^{1/3} n}\right] \le \exp\left(-\frac{cn^{2/3}}{\log^{1/3} n}\right) = o(n^{-c}).$$

Similarly, for each value of k, we have $\mathbb{E}[|S_k|] = \frac{8c^2 n^{2/3}\log^{2/3} n}{k}$, and

$$\Pr\left[|S_k| > \frac{32c^2 n^{2/3}\log^{2/3} n}{k}\right] \le \exp\left(-\frac{8c^2 n^{2/3}\log^{2/3} n}{k}\right) = O(n^{-c}),$$

where the last equality follows since $k \le \frac{n^{2/3}}{\log^{1/3} n}$. A union bound implies that $|\mathcal{C}| = O\left(\frac{n^{2/3}}{\log^{1/3} n}\right)$ and $|S_k| = O\left(\frac{n^{2/3}\log^{2/3} n}{k}\right)$ for all k, w.p. at least $1 - O(n^{-c+1})$.

Finally, for each k, for each $(c_i, c_j) \in \mathcal{C} \times S_k$ we add at most one path with less than $2k$ missing edges to H. Thus, for each value of k we add less than $|\mathcal{C}| \cdot |S_k| \cdot 2k = O(n^{4/3}\log^{1/3} n)$ edges to H, w.p. at least $1 - O(n^{-c+1})$. Summing over all $O(\log n)$ values of k, and adding the number of edges contributed by the clustering phase, we conclude that H has at most $O(n^{4/3}\log^{4/3} n)$ edges, w.p. at least $1 - O(n^{-c+1})$. ◀

▶ **Lemma 10.** *The graph H constructed by Algorithm* `6AP` *satisfies $\delta_H(x, y) \le \delta_G(x, y) + 6$ for each pair $(x, y) \in V \times V$, with probability at least $1 - O\left(n^{-c+2}\right)$.*

Proof. Consider a shortest path ρ in G between two nodes $x, y \in V$ (see Figure 1). Let x' and y' be the first and last clustered nodes on ρ, respectively. If all nodes of ρ are unclustered, then ρ is fully contained in H_0 and we are done.

Let c_1 and c_3 be the centers of the clusters containing x' and y', respectively. Let σ be a shortest path in G between c_1 and c_3, and denote by k' the number of edges of $\sigma \setminus H_0$. Let k be the largest power of 2 such that $k \le k'$.

An edge can be in $\sigma \setminus H_0$ only if it connects two clustered nodes. Hence, k', the number of edges in $\sigma \setminus H_0$, is smaller than the number of clustered nodes in σ. On the other hand, σ cannot contain more than three nodes of the same cluster: the distance between every two nodes in a cluster is at most two, so a shortest path cannot traverse more than three nodes of the same cluster. Thus, the number of clusters intersecting σ is at least $k'/3$. As $k'/3 \ge k/3$, the probability that none of the centers of these clusters is chosen to S_k is at most $\left(1 - \frac{8c^2\log n}{k}\right)^{k/3} = O\left(n^{-c^2}\right)$. For each pair of nodes, a cluster center on a shortest path between them is chosen to S_k, for the appropriate value of k, with similar probability. A union bound implies that this claim holds for all pairs in $V \times V$ w.p. at least $1 - O(n^{-c^2+2})$.

Let w be a node on σ in a cluster C_2 such that $c_2 \in S_k$, if such a cluster exists. Denote by $\sigma[c_1, w]$ the sub-path of σ from c_1 to w. As there are $k' < 2k$ edges in $\sigma \setminus H_0$, there are also less than $2k$ edges in $\sigma[c_1, w] \setminus H_0$. Thus, in step $3(b)$ of the path-buying phase for k, either the path $\sigma[c_1, w]$ or some other path between c_1 and w of length at most $\delta_G(c_1, w)$ is added to A. In step $3(c)$, a path from c_1 to some node $w_1 \in C_2$ is added to H, and this is a shortest path in A, so $\delta_H(c_1, w_1) \le \delta_G(c_1, w)$. Similarly, a shortest path from c_3 to some $w_3 \in C_2$ is added to H, and $\delta_H(c_3, w_3) \le \delta_G(c_3, w)$.

The path σ is a shortest path from c_1 to c_3 in G, so $|\sigma| \le \delta_G(x', y') + 2$. As $\delta_G(c_1, w) + \delta_G(c_3, w) = |\sigma|$, we conclude $\delta_H(c_1, w_1) + \delta_H(c_3, w_3) \le |\sigma| \le \delta_G(x', y') + 2$.

Consider the path from x to y in H composed of the sub-path of ρ from x to x', the edge (x', c_1), the path from c_1 to w_1, the edges (w_1, c_2) and (c_2, w_3), the path from w_3 to c_3, the edge (c_3, y'), and finally, the sub-path of ρ from y' to y. This is a path from x to y in H, implying

$$\begin{aligned}\delta_H(x, y) &\le \delta_H(x, x') + 1 + \delta_H(c_1, w_1) + 2 + \delta_H(w_3, c_3) + 1 + \delta_H(y', y)\\ &\le \delta_G(x, x') + 4 + \delta_G(x', y') + 2 + \delta_G(y', y) = \delta_G(x, y) + 6,\end{aligned}$$

as desired. ◀

We now discuss the implementation of Algorithm `6AP` in the CONGEST model.

▶ **Lemma 11.** *Algorithm* `6AP` *can be implemented in* $O\left(\frac{n^{2/3}}{\log^{1/3} n} + D\right)$ *rounds in the* CONGEST *model, with probability at least* $1 - o(n^{-c})$.

Proof. For the clustering phase, each node decides locally w.p. $\frac{c}{n^{1/3}\log^{1/3} n}$ to become a cluster center, and notifies its neighbors. Each node with a neighbor that is a cluster center now joins a cluster by sending a message to such a neighbor and adding the approprеate edge to the spanner. A node with no neighboring cluster centers notifies all its neighbors and adds all its edges to the spanner. This is done in a constant number of rounds.

Before the path buying phase, the nodes construct a single BFS tree, along which they compute an upper bound D' on D, satisfying $D \le D' < 2D$, and count the number of cluster centers, $|\mathcal{C}|$. The nodes mark the edges of H_0 with weight 0 and the other edges with weight 1. Then, they construct a WBFS tree rooted at each cluster center by executing Algorithm 1 for $|\mathcal{C}| + D'$ many rounds. By the proof of Lemma 9, we have $|\mathcal{C}| \in O\left(\frac{n^{2/3}}{\log^{1/3} n}\right)$ w.p. at least $1 - o(n^{-c})$, and thus the construction of the WBFS trees takes $O\left(\frac{n^{2/3}}{\log^{1/3} n} + D\right)$ rounds with the same probability.

Each node v now knows about a "good" path to each cluster center c_i, i.e., a shortest path from c_i to v, with a minimal number of edges not in H after the clustering phase. A node v in a cluster C_j notifies its neighbor c_j about all the distances to other cluster centers in $\mathcal{C}$ and the number of missing edges in each such path. That is, each $v \in C_j$ sends $|\mathcal{C}|$ messages to c_j, which takes $O\left(\frac{n^{2/3}}{\log^{1/3} n}\right)$ rounds.

Each cluster center c_j decides locally to join each set S_k w.p. $\frac{8c^2 \log n}{k}$. For each other center $c_i \in \mathcal{C}$, c_j locally constructs the list A: for each $v \in C_j$, A contains the shortest path from c_i to $v \in C_j$ found by the WBFS algorithm, and the number of missing edges in it. Then, c_j chooses from A a path from c_i to some $v \in C_j$ with a minimal number of missing edges, and if it has at most $2k$ missing edges, c_j sends a "buy c_i" message to v.

Finally, all nodes simultaneously execute a "buy" phase, where "buy c_i" messages are sent up the WBFS tree. To avoid congestion, we assume that during the execution of Algorithm 1, each node keeps a record of the messages it got in each round and the WBFS source each

message referred to. Each node v then sends messages in reversed order: if v has a message "buy c_i", and it got a message from u regarding c_i in the r-before-last round of Algorithm 1, then it sends the message "buy c_i" to u in round r of the "buy" phase. Then, u adds "buy c_i" to its list of messages, and adds the edge (u, v) to the spanner. This parts takes $O\left(\frac{n^{2/3}}{\log^{1/3} n} + D\right)$ rounds, just like the execution of Algorithm 1. ◀

5 Discussion and Open Questions

While we present an application of WBFS trees, our algorithm also solves the weighted (S, d, k)-detection problem, a result the could be of independent interest.

The question of finding the *lightest* paths between all pairs of nodes in a graph is a fundamental question in many computational models. In the CONGEST model, no exact algorithm for the problem running in $O(n)$ rounds is known. This major question is still left open here, but we hope our study of lightest shortest paths could facilitate future research on the question of finding lightest paths.

While this paper settles the question of constructing sparse (+6)-spanners fast, the study of spanner construction in distributed environments still lags behind the study of sequential spanner construction algorithms. In the field of purely additive spanners, we still do not have fast algorithms, e.g., for the construction of sparse (+0)-pairwise spanners (a.k.a. pairwise preservers) and (+6)-pairwise spanners.

A more intriguing question is proving time lower bounds for the construction of spanners in the CONGEST model: while $\Omega(D)$ rounds are known to be necessary [44], lower bounds that depend on other parameters of the graph or the spanners exist only for pairwise spanners [10]. Finding a lower bound for the construction of all-pairs spanners in the CONGEST model is still an open question. Such a lower bound could show that the $n^{3/2}$ term in the time bound of our construction is inevitable, or motivate a design of faster algorithms for the problem.

References

1. Amir Abboud and Greg Bodwin. Error Amplification for Pairwise Spanner Lower Bounds. In *27th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 841–854, 2016.
2. Amir Abboud and Greg Bodwin. The 4/3 Additive Spanner Exponent Is Tight. *J. ACM*, 64(4):28:1–28:20, 2017.
3. Amir Abboud, Keren Censor-Hillel, and Seri Khoury. Near-Linear Lower Bounds for Distributed Distance Computations, Even in Sparse Networks. In *30th International Symposium on Distributed Computing, DISC*, pages 29–42, 2016.
4. 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 *ACM Symposium on Principles of Distributed Computing, PODC*, pages 199–205, 2018.
5. Baruch Awerbuch. Optimal Distributed Algorithms for Minimum Weight Spanning Tree, Counting, Leader Election and Related Problems. In *ACM Symposium on Theory of Computing, STOC*, pages 230–240, 1987.
6. Surender Baswana, Telikepalli Kavitha, Kurt Mehlhorn, and Seth Pettie. Additive spanners and (alpha, beta)-spanners. *ACM Trans. Algorithms*, 7(1):5, 2010.
7. Surender Baswana and Sandeep Sen. A simple and linear time randomized algorithm for computing sparse spanners in weighted graphs. *Random Struct. Algorithms*, 30(4):532–563, 2007.

8 Keren Censor-Hillel and Michal Dory. Distributed Spanner Approximation. In *ACM Symposium on Principles of Distributed Computing, PODC*, pages 139–148, 2018.

9 Keren Censor-Hillel, Bernhard Haeupler, Jonathan A. Kelner, and Petar Maymounkov. Global computation in a poorly connected world: fast rumor spreading with no dependence on conductance. In *44th Symposium on Theory of Computing Conference, STOC*, pages 961–970, 2012.

10 Keren Censor-Hillel, Telikepalli Kavitha, Ami Paz, and Amir Yehudayoff. Distributed Construction of Purely Additive Spanners. In *30th International Symposium on Distributed Computing, DISC*, pages 129–142, 2016.

11 Keren Censor-Hillel, Ami Paz, and Noam Ravid. The Sparsest Additive Spanner via Multiple Weighted BFS Trees. *CoRR*, abs/1811.01997, 2018. arXiv:1811.01997.

12 Shiri Chechik. Compact routing schemes with improved stretch. In *ACM Symposium on Principles of Distributed Computing, PODC*, pages 33–41, 2013.

13 Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. Distributed Verification and Hardness of Distributed Approximation. *SIAM J. Comput.*, 41(5):1235–1265, 2012.

14 Bilel Derbel and Cyril Gavoille. Fast deterministic distributed algorithms for sparse spanners. *Theor. Comput. Sci.*, 399(1-2):83–100, 2008.

15 Bilel Derbel, Cyril Gavoille, and David Peleg. Deterministic Distributed Construction of Linear Stretch Spanners in Polylogarithmic Time. In *21st International Symposium on Distributed Computing, DISC*, pages 179–192, 2007.

16 Bilel Derbel, Cyril Gavoille, David Peleg, and Laurent Viennot. On the locality of distributed sparse spanner construction. In *27th Annual ACM Symposium on Principles of Distributed Computing, PODC*, pages 273–282, 2008.

17 Bilel Derbel, Cyril Gavoille, David Peleg, and Laurent Viennot. Local Computation of Nearly Additive Spanners. In *International Symposium on Distributed Computing, DISC*, pages 176–190, 2009.

18 Devdatt P. Dubhashi, Alessandro Mei, Alessandro Panconesi, Jaikumar Radhakrishnan, and Aravind Srinivasan. Fast distributed algorithms for (weakly) connected dominating sets and linear-size skeletons. *J. Comput. Syst. Sci.*, 71(4):467–479, 2005.

19 Michael Elkin. Computing almost shortest paths. *ACM Trans. Algorithms*, 1(2):283–323, 2005.

20 Michael Elkin. Distributed exact shortest paths in sublinear time. In *ACM SIGACT Symposium on Theory of Computing, STOC*, pages 757–770, 2017.

21 Michael Elkin and Ofer Neiman. Hopsets with Constant Hopbound, and Applications to Approximate Shortest Paths. In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS*, pages 128–137, 2016.

22 Michael Elkin and Jian Zhang. Efficient algorithms for constructing $(1+\epsilon,\beta)$-spanners in the distributed and streaming models. *Distributed Computing*, 18(5):375–385, 2006.

23 Robert G. Gallager, Pierre A. Humblet, and Philip M. Spira. A Distributed Algorithm for Minimum-Weight Spanning Trees. *ACM Trans. Program. Lang. Syst.*, 5(1):66–77, 1983.

24 Mohsen Ghaffari and Fabian Kuhn. Derandomizing Distributed Algorithms with Small Messages: Spanners and Dominating Set. In *International Symposium on Distributed Computing, DISC*, volume 121 of *LIPIcs*, pages 29:1–29:17, 2018.

25 Mohsen Ghaffari and Jason Li. Improved distributed algorithms for exact shortest paths. In *SIGACT Symposium on Theory of Computing, STOC*, pages 431–444, 2018.

26 Ofer Grossman and Merav Parter. Improved Deterministic Distributed Construction of Spanners. In *31st International Symposium on Distributed Computing, DISC*, pages 24:1–24:16, 2017.

27 Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. A deterministic almost-tight distributed algorithm for approximating single-source shortest paths. In *ACM SIGACT Symposium on Theory of Computing, STOC*, pages 489–498, 2016.

28 Stephan Holzer. Personal communication.

29 Stephan Holzer, David Peleg, Liam Roditty, and Roger Wattenhofer. Distributed 3/2-Approximation of the Diameter. In *28th International Symposium on Distributed Computing, DISC*, pages 562–564, 2014.

30 Stephan Holzer and Roger Wattenhofer. Optimal distributed all pairs shortest paths and applications. In *ACM Symposium on Principles of Distributed Computing, PODC*, pages 355–364, 2012.

31 Chien-Chung Huang, Danupon Nanongkai, and Thatchaphol Saranurak. Distributed Exact Weighted All-Pairs Shortest Paths in $\tilde{O}(n^{5/4})$ Rounds. In *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS*, pages 168–179, 2017.

32 Telikepalli Kavitha. New Pairwise Spanners. In *32nd International Symposium on Theoretical Aspects of Computer Science, STACS*, pages 513–526, 2015.

33 Mathias Bæk Tejs Knudsen. Additive Spanners: A Simple Construction. In *14th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT*, pages 277–281, 2014.

34 Christoph Lenzen, Boaz Patt-Shamir, and David Peleg. Distributed distance computation and routing with small messages. *Distributed Computing*, pages 1–25, 2018.

35 Christoph Lenzen and David Peleg. Efficient distributed source detection with limited bandwidth. In *ACM Symposium on Principles of Distributed Computing, PODC*, pages 375–382, 2013.

36 Zvi Lotker, Boaz Patt-Shamir, and Seth Pettie. Improved Distributed Approximate Matching. *J. ACM*, 62(5):38:1–38:17, 2015.

37 Danupon Nanongkai. Distributed approximation algorithms for weighted shortest paths. In *Symposium on Theory of Computing, STOC*, pages 565–573, 2014.

38 Merav Parter. Vertex fault tolerant additive spanners. *Distributed Computing*, 30(5):357–372, 2017.

39 Merav Parter and Eylon Yogev. Congested Clique Algorithms for Graph Spanners. In *International Symposium on Distributed Computing, DISC*, volume 121 of *LIPIcs*, pages 40:1–40:18, 2018.

40 David Peleg. *Distributed Computing: A Locality-Sensitive Approach.* Monographs on Discrete Mathematics and Applications. Society for Industrial and Applied Mathematics, 2000.

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

42 David Peleg and Jeffrey D. Ullman. An Optimal Synchronizer for the Hypercube. *SIAM J. Comput.*, 18(4):740–747, 1989.

43 David Peleg and Eli Upfal. A trade-off between space and efficiency for routing tables. *J. ACM*, 36(3):510–530, 1989.

44 Seth Pettie. Distributed algorithms for ultrasparse spanners and linear size skeletons. *Distributed Computing*, 22(3):147–166, 2010.

45 Mikkel Thorup and Uri Zwick. Compact routing schemes. In *ACM Symposium on Parallelism in Algorithms and Architectures, SPAA*, pages 1–10, 2001.

46 David P. Woodruff. Additive Spanners in Nearly Quadratic Time. In *37th International Colloquium on Automata, Languages and Programming, ICALP*, pages 463–474, 2010.

The Amortized Analysis of a Non-blocking Chromatic Tree

Jeremy Ko
Department of Computer Science, University of Toronto, Canada
jerko@cs.toronto.edu

Abstract

A non-blocking chromatic tree is a type of balanced binary search tree where multiple processes can concurrently perform search and update operations. We prove that a certain implementation has amortized cost $O(\dot{c}+\log n)$ for each operation, where $\dot{c}$ is the maximum number of concurrent operations at any point during the execution and n is the maximum number of keys in the tree during the operation. This amortized analysis presents new challenges compared to existing analyses of other non-blocking data structures.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis, Theory of computation → Distributed algorithms

Keywords and phrases amortized analysis, non-blocking, lock-free, balanced binary search trees

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.8

Related Version A full version is available at `https://arxiv.org/abs/1811.06383`.

Funding Funding was provided by the Natural Sciences and Engineering Research Council of Canada.

Acknowledgements I want to thank my supervisor, Faith Ellen, whose guidance and feedback was very helpful in writing this paper. I would also like to thank Eric Ruppert and the anonymous reviewers.

1 Introduction

A *concurrent data structure* is one that can be concurrently updated by multiple processes. A *wait-free* implementation of a concurrent data structure guarantees that every operation completes within a finite number of steps by the process that invoked the operation. A *non-blocking* implementation of a concurrent data structure guarantees that whenever there are active operations, one operation will eventually complete in a finite number of steps. Since a particular operation may not complete in a finite number of steps, it not possible to perform a worst-case analysis for non-blocking data structures. However, such implementations are desirable since they are typically less complicated than wait-free implementations and often perform well in practice. Amortized analysis gives an upper bound on the worst-case number of steps performed during a sequence of operations in an execution, rather than on the worst-case step complexity of a single operation. This type of analysis is useful for expressing the efficiency of non-blocking data structures.

In this paper, we present an amortized analysis of a non-blocking chromatic tree. The chromatic tree was introduced by Nurmi and Soisalon-Soininen [12] as a generalization of a red-black tree with relaxed balance conditions, allowing insertions and deletions to be performed independently of rebalancing. Boyar, Fagerberg, and Larsen [2] showed that the amortized number of rebalancing transformations per update is constant, provided each

© Jeremy Ko;
licensed under Creative Commons License CC-BY

update and each rebalancing transformation is performed without interference from other operations.

Brown, Ellen, and Ruppert [5] gave a non-blocking implementation using LLX and SCX primitives [4], which themselves can be implemented using CAS. Unfortunately, this implementation does not have good amortized complexity. We prove that a slightly modified implementation has amortized cost $O(\dot{c}(\alpha) + \log n(op))$ for each operation *op* in an execution α, where $\dot{c}(\alpha)$ is the point contention of α and $n(op)$ is the maximum number of nodes in the chromatic tree during *op*'s execution interval.

Our amortized analysis for the chromatic tree is based on the amortized analysis for the unbalanced binary search tree by Ellen, Fatourou, Helga, and Ruppert [7]. Several challenges make our analysis more difficult than their analysis.

The unbalanced binary search tree only has one transformation to perform insertions and one transformation to perform deletions. The chromatic tree also has 11 different rebalancing transformations (not including their mirror images). Our analysis has the feature that it does not require a separate case for each type of transformation performed. A very similar analysis should give the same amortized step complexity for other types of balanced binary search trees, especially for those implemented using LLX and SCX [3].

Unlike update transformations, which only occur at leaf nodes, rebalancing transformations may occur anywhere in the tree. Furthermore, operations may perform rebalancing transformations on behalf of other operations, or may cause them to perform additional rebalancing transformations they would otherwise not perform. For an amortized analysis done using the accounting method, these facts make it much more difficult to determine the steps in which dollars should be deposited into bank accounts to pay for future steps.

Much like the analysis of the unbalanced binary search tree, many overlapping transformations can cause all but one to fail. Many operations may also be working together to perform the same rebalancing transformations. The amortized analysis of the chromatic tree is complicated by the fact that the constant upper bound on the number of rebalancing transformations per operation is amortized, even when each update and rebalancing transformation is performed atomically. Since the number of rebalancing transformations performed by each operation may differ, a bad configuration in which many rebalancing transformations can occur may also have high contention. Executions including such configurations must be accounted for in the amortized analysis.

The remainder of this paper is organized as follows. In Section 2, we present the asynchronous shared memory model assumed in our analysis. In Section 3, we give an overview of related work done on the amortized analysis of concurrent data structures. In Section 4, we give an overview of the modified chromatic tree implementation. Finally, in Section 5, we determine its amortized step complexity.

2 Model

Throughout this paper, we use an asynchronous shared memory model [1]. Shared memory consists of a collection of shared variables accessible by all processes in a system. Processes communicate using read, write, and CAS primitives.

A *configuration* of a system consists of the values of all shared variables and the states of all processes. A *step* by a process applies a primitive to a shared variable and can also change the state of the process. An *execution* is an alternating sequence of configurations and steps, starting with a configuration. A *solo execution* from a configuration C by a process P is an execution starting from C in which all steps are performed by P.

An *abstract data type* is a collection of mathematical objects and operations that satisfies certain properties. A *concurrent data structure* for the abstract data type provides representations of the objects in shared memory and algorithms for the processes to perform the operations. An operation on a data structure by a process becomes *active* when the process performs the first step of its algorithm. The operation becomes *inactive* after the last step of the algorithm is performed by the process. The *execution interval* of the operation consists of all configurations in which it is active. In the initial configuration, the data structure represents an empty instance of the abstract data type and there are no active operations.

The *worst-case step complexity* of an operation is the maximum number of steps taken by a process during the execution interval of any instance of this operation in any execution. The *amortized step complexity* of a data structure is the maximum number of steps in any execution consisting of operations on the data structure, divided by the number operations invoked in the execution. One can determine an upper bound on the amortized step complexity by assigning an amortized cost to each operation, such that for all possible executions α on the data structure, the total number of steps taken in α is at most the sum of the amortized costs of the operations in α. The amortized cost of a concurrent operation is often expressed as a function of contention. For a configuration C, we define its *contention* $\dot{c}(C)$ to be the number of active operations in C. For an operation op in an execution α, we define its *point contention* $\dot{c}(op)$ to be the maximum number of active operations in a single configuration during the execution interval of op. Finally, for an execution α, we define its point contention $\dot{c}(\alpha)$ to be the maximum number of active operations in a single configuration during α.

3 Related Amortized Analyses of Non-blocking Data Structures

In this section, we give an overview of the amortized analyses of various non-blocking data structures. While there are many papers that give implementations of non-blocking data structures, there are relatively few that give the amortized step complexity of their implementations.

Fomitchev and Ruppert [9] modify an implementation of a singly linked list by Harris [11] with the addition of *backlinks*. A backlink is a pointer to the predecessor of a node that has been *marked* for deletion. Whenever an update operation fails, the backlinks can be followed until an unmarked node is reached, rather than restarting searches from the head of the linked list. They prove an amortized step complexity of $O(n(op) + \dot{c}(op))$ for their improved implementation, where $n(op)$ is the number of elements in the linked list when operation op is invoked. The amortized analysis is done by charging each failed CAS and each backlink traversal performed by an operation op to a successful CAS of a concurrent operation in the execution interval of op. Gibson and Gramoli [10] give a simplified implementation of a linked list and claim it has the same amortized step complexity.

Ellen, Fatourou, Ruppert, and van Breugel [8] give the first provably correct non-blocking implementation of an unbalanced binary search tree using CAS primitives. Ellen, Fatourou, Helga, and Ruppert [7] show that this implementation has poor amortized step complexity. They improve the implementation by allowing failed update attempts to backtrack to a suitable internal node in the tree, rather than restarting attempts from the root. They prove for their implementation, each operation op has amortized cost $O(h(op) + \dot{c}(op))$, where $h(op)$ is the height of the binary search tree when op is invoked. The amortized analysis is difficult because a single DELETE operation can cause a sequence of overlapping DELETE operations by other processes to fail. The accounting method is used to show how all failed attempts in an execution can be paid for. Chatterjee, Nguyen and Tsigas [6] give a different

implementation of an unbalanced binary search tree, and give a sketch that the amortized cost of each operation in their implementation is also $O(h(op) + \dot{c}(op))$.

Shafiei [13] gives an implementation of a non-blocking doubly-linked list. She proves an amortized cost of $O(\dot{c}(op))$ for update operations *op*. The analysis closely follows the analysis of the unbalanced binary search tree [7], except modified to use the potential method.

4 The Non-blocking Implementation of the Chromatic Tree

A chromatic tree is a data structure for a dynamic set of elements, each with a distinct key from an ordered universe. It supports the following three operations:

- INSERT(k), which adds an element with key k into the set and returns TRUE if the set does not contain an element with key k; otherwise it returns FALSE,
- DELETE(k), which removes the element with key k from the set and returns TRUE if there is such an element; otherwise it returns FALSE, and
- FIND(k), which returns TRUE if there is an element in the set with key k and FALSE otherwise.

A chromatic tree is a generalization of a red-black tree with relaxed balance conditions. It is *leaf-oriented*, so every element in the dynamic set represented by the data structure corresponds to a leaf and all nodes have 0 or 2 children. Each node in a chromatic tree has a non-negative weight. We call a node *red* if it has weight 0, *black* if it has weight 1, and *overweight* if it has weight greater than 1. The *weighted level* of a node x is the sum of node weights along the path from the root node to x. The balance conditions of red-black trees and chromatic trees as described in [2] are as follows.

▶ **Definition 1.** A *red-black tree* T satisfies following balance conditions:

B1. The leaves of T are black.

B2. All leaves of T have the same weighted level.

B3. No path from T's root to a leaf contains two consecutive red nodes.

B4. T has only red and black nodes.

▶ **Definition 2.** A *chromatic tree* T satisfies the following balance conditions:

C1. The leaves of T are not red.

C2. All leaves of T have the same weighted level.

Let x be a node that is in the chromatic tree with weight $x.w$. If x and its parent both have weight 0, then a *red-red violation* occurs at x. If $x.w > 1$, then $x.w - 1$ *overweight violations* occur at x. When a chromatic tree contains no violations, it satisfies the balance conditions of a red-black tree. The balance conditions of a red-black tree ensure its height is $O(\log n)$, where n is the number of nodes in the tree.

An example of a chromatic tree is shown in Figure 1. The number inside each node is its key and the number to its right is its weight. The weighted level of each leaf is 3. There is an overweight violation at the leaf with key 1, and a red-red violation at the internal node with key 6.

Rebalancing is done independently of insertions and deletions. This allows rebalancing to be done at times when there are fewer processes that want to perform operations. However, the chromatic tree may not be height balanced at all times. The following lemma proven by Boyar, Fagerberg, and Larsen [2] gives an upper bound on the total number of rebalancing transformations that can occur.

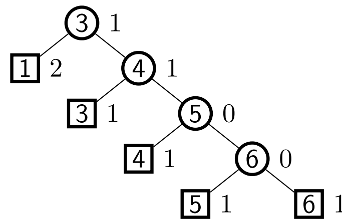


Figure 1 A chromatic tree containing elements with keys 1, 3, 4, 5 and 6.

▶ **Lemma 3.** *If $i > 0$ insertions and d deletions are performed on an initially empty chromatic tree, then at most $3i + d - 2$ rebalancing transformations can occur.*

In the non-blocking implementation by Brown, Ellen, and Ruppert [5], rebalancing is done as a part of Insert and Delete operations. They proved that the imbalance in the chromatic tree depends on the number of active operations.

▶ **Lemma 4.** *If there are c active* Insert *and* Delete *operations and the chromatic tree contains n elements, then its height is $O(c + \log n)$.*

4.1 An Implementation of a Chromatic Tree using LLX and SCX

An implementation of a chromatic tree was first given by Brown, Ellen, and Ruppert [5] using the LLX and SCX primitives. We describe a slightly modified implementation with better amortized step complexity. LLX and SCX are themselves implemented using CAS primitives. It is necessary to understand the step complexity of LLX and SCX to analyze the chromatic tree.

LLX and SCX operate on Data-records. A Data-record is a collection of fields used to represent a natural piece of a data structure. A successful LLX(r) on Data-record r returns a snapshot of the fields of r. A process can only perform SCX(V, R, fld, new) if it has previously performed a successful LLX on each Data-record in V since it last performed SCX (or the beginning of the execution if it has never performed SCX). A successful SCX(V, R, fld, new) *finalizes* all Data-records in $R \subseteq V$. It also updates fld, one field of a single Data-record in V, to the value new. This removes the Data-records in R from the data structure. An unsuccessful SCX returns False, which only occurs when there is a concurrent SCX(V', R', fld', new') such that $V \cap V' \neq \emptyset$. Likewise an LLX can return Fail if there is a concurrent SCX(V', R', fld', new') such that $r \in V'$. An LLX(r) returns Finalized if r has been previously finalized by an SCX. In both of these cases, the LLX is unsuccessful.

Each *node* of the chromatic tree is represented by one Data-record that contains fields for its key, weight, child pointers, *marked* bit, and *info* pointer. The marked bit is used to finalize nodes. A node is only removed from the chromatic tree after it is marked. Once a marked bit is set to True, it cannot be changed back to False. The info pointer points to an SCX-record, which contains information required for processes to perform a pending SCX on behalf of another process. It contains a *state* field whose value is either InProgress, Aborted, or Committed, and is initially InProgress.

Consider an SCX(V, R, fld, new) performed by a process P. For the chromatic tree, each node of R is reachable from the node pointed to by fld by only following child pointers of nodes in R. P first creates a new SCX-record U for this SCX. A node r is *frozen* for U if $r.info$ points to U and either $U.state$ = InProgress, or $U.state$ = Committed and r is marked. Freezing acts like a lock on a node held by a particular operation. For each node

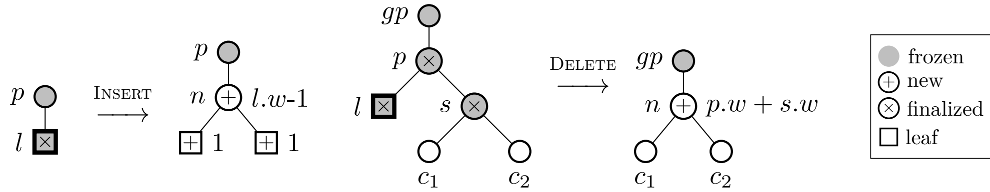


Figure 2 The update transformations of the chromatic tree.

$v \in V$, P attempts to freeze v by setting $v.info$ to point to U using a *freezing* CAS. This only succeeds if v has not been frozen since P's last LLX on v. If the freezing CAS fails and no other process has frozen v for U, then P performs an *abort step* and returns False. An abort step atomically unfreezes all nodes frozen for U by setting U to the Aborted state.

Once all nodes in V are successfully frozen for U, the SCX can no longer fail. Then each node in $R \subseteq V$ is *marked* for removal and an *update* CAS sets fld to the value new. At this point, the nodes in R are no longer reachable from the root of the chromatic tree. Finally, a *commit step* is performed, which atomically unfreezes all nodes in $V \setminus R$ by setting U to the Committed state. The nodes in R remain frozen.

Consider an LLX(r) by a process P. If r is marked when the LLX is invoked, P will help the SCX that marked r (if it is not yet complete) and return Finalized. Otherwise P attempts to take a snapshot of the fields of r. If r is not frozen while P reads each of its fields, the LLX is successful and returns the fields of r. If a concurrent SCX operation freezes r sometime during P's LLX, P may help the concurrent SCX complete before returning Fail.

LLX and SCX guarantee non-blocking progress. If SCX is performed infinitely often, then an infinite number succeed [4]. Much of our amortized analysis will require a mechanism to charge a failed LLX or SCX of one operation to a different operation that caused it to fail.

We give an overview of the algorithm to Insert an element with key k into a chromatic tree. The algorithm for the Delete operation is similar. The Insert operation is divided into an *update phase* and a *cleanup phase*. An update phase repeatedly performs update attempts until one is successful.

An update attempt begins with a search through the tree for the key k, starting from the root, until it visits a leaf l. Let p denote the node visited immediately before visiting l. If $l.key = k$, then the key is already in the chromatic tree, so Insert returns False. Otherwise a subroutine TryInsert(p, l) is invoked, which attempts to perform an LLX on the nodes p and l, followed by a single SCX to apply the Insert transformation shown in Figure 2. Gray nodes represent nodes on which LLXs and freezing CASs are performed. Nodes marked with $\times$ are removed from the chromatic tree, while nodes marked with $+$ are new nodes added to the tree. Any weight restrictions on a node are shown to its right. Notice that, to complete the transformation, an SCX is only required to update the pointer of a single node to point to a subtree of new nodes. The instance of TryInsert fails if any of its LLXs or SCXs are unsuccessful, in which case a new update attempt begins immediately following the failed LLX or SCX. If a violation is created during the insertion performed by a successful TryInsert, the operation enters its cleanup phase. Otherwise, the Insert terminates.

Ellen, Fatourou, and Helga, and Ruppert [7] use a stack to recover from failed update attempts in their unbalanced binary search tree. Each time a process visits a node during its search, it pushes a pointer to this node onto a local stack. After a failed attempt, the process can pop nodes off its stack until an unmarked node is found. This allows the process

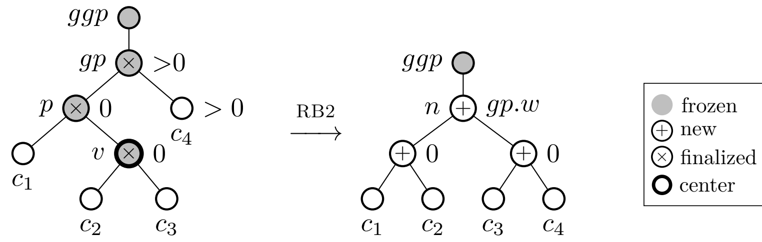


Figure 3 One of the 11 different rebalancing transformations.

to recover from a failed attempt without restarting its search from the root. A process is *backtracking* during the interval in which it pops nodes off its stack. As described in [7], update operations that restart searches from the root without using a stack have poor amortized cost. Consequently, we will consider an implementation of the chromatic tree using a stack to recover from failed attempts. The proof of correctness of this modification appears in the full version.

The cleanup phase is also divided into a series of attempts. At the start of the cleanup phase, the process empties its local stack and performs a new search for k starting from the root. When a new attempt begins, the process pops nodes off its stack until an unmarked node is found. Then it continues searching for k from this node. The cleanup phase terminates when the search reaches a leaf without encountering a violation. If a violation is encountered at a node v during the search, a subroutine TryRebalance(ggp, gp, p, v) attempts to remove the violation, where p, gp, and ggp are the last 3 nodes visited on the search to v. TryRebalance determines which of the 11 rebalancing transformations should be applied. It applies the rebalancing transformation by performing LLX on each node involved in the transformation and then using SCX to update one pointer. LLXs are performed from top to bottom. If two nodes involved in the transformation have the same height, then one of these is ggp, gp, p, or v, and an LLX is performed on this node first. If any LLX or SCX is unsuccessful, TryRebalance immediately returns. Whether or not TryRebalance successfully performs a rebalancing transformation, a new attempt begins. A cleanup attempt is successful if it performs a successful SCX during TryRebalance; otherwise it is unsuccessful.

For our analysis, we identify a single node for each rebalancing transformation as its *center*. In Figure 3, the node with a bold outline is the center node. The center node of a transformation always contains a violation. A rebalancing transformation is *centered* at a violation if the violation occurs at the center node of the transformation. We classify each node involved in an SCX as either a downwards node or a cross node.

▶ **Definition 5.** Consider an instance S of SCX(V, R, fld, new), where $V = \{v_1, \ldots, v_k\}$, enumerated in the order they are frozen. For $1 \leq i < k$, v_i is a *cross node* for S if v_i is the sibling of v_{i+1}, otherwise v_i is a *downwards node* for S. We define v_k to be a downwards node for S.

Only Insert transformations create red-red violations and only Delete transformations create overweight violations. For the Insert transformation shown in Figure 2, if $p.w = 0$ and $l.w = 1$, then a red-red violation is created at the new node n. The Delete transformation creates an overweight violation at the new node n if $p.w > 0$ and $s.w > 0$, and moves any overweight violations from p or s to n.

Rebalancing transformations remove at most 1 violation. Consider the rebalancing transformation RB2 shown in Figure 3. The red-red violation located at the center node v is removed. If $gp.w > 1$, then there are $gp.w - 1$ overweight violations at gp. We say that these overweight violations are moved to the new node n. Similar definitions can be made for the remaining 10 rebalancing transformations.

Some rebalancing transformations may move violations, without removing any violations. Thus, several rebalancing transformations may be centered at a violation before it is eventually removed.

▶ **Definition 6.** If the update phase of an operation creates a violation and, hence, begins a cleanup phase, cp, let $viol(cp)$ denote this violation. Let $rebal(viol(cp))$ be the number of times during the execution $viol(cp)$ is located at the center of a successful rebalancing transformation.

Intuitively, our analysis will charge cp for every successful rebalancing transformation centered at $viol(cp)$.

We can show that cp does not terminate until $viol(cp)$ is removed. Since an update transformation creates at most 1 violation, the number of violations in the chromatic tree is bounded by the number of incomplete Insert and Delete operations. This fact is used to show that Lemma 4 continues to hold when backtracking is used.

5 Chromatic Tree Amortized Analysis

In this section, we present an amortized step complexity of the chromatic tree. We use the following notation.

▶ **Definition 7.** For each configuration C and operation op with update phase up and cleanup phase cp of an execution α, let:

- $h(C)$ be the height of the chromatic tree in configuration C,
- $h(up)$ and $h(cp)$ be the height of the chromatic tree in the starting configurations of up and cp, respectively,
- $h(op)$ be the maximum height of the chromatic tree over all configurations in the execution interval of op, and
- $\dot{c}(up)$ and $\dot{c}(cp)$ be the maximum number of active operations in a single configuration in the execution interval of up and cp respectively.

We first give a sketch of our amortized analysis. Its structure is similar to the amortized analysis of a binary search tree by Ellen et al [7].

▶ **Definition 8.** For an update phase or cleanup phase xp, let $attempts(xp)$ be the number of attempts made by xp, and $pushes(xp)$ be the number of times xp pushes a node onto its stack.

The number of steps an operation takes during searches is a constant times the number of pushes it performs. Each attempt performs a constant number of LLXs and SCXs, each of which takes a constant number of steps. These observations imply the following result.

▶ **Lemma 9.** *The number of steps taken by an operation op with update phase up and cleanup phase cp is* $steps(op) = O(attempts(up) + attempts(cp) + pushes(up) + pushes(cp))$. *If op has no cleanup phase, then* $steps(op) = O(attempts(up) + pushes(up))$.

In the amortized analysis of the binary search tree [7], it is shown how the total number of pushes can be bounded by a function of the total number of attempts, the height of the tree at the beginning of each operation, and point contention. For the chromatic tree, we similarly prove that for any finite execution α,

$$\sum_{up \in \alpha} \mathit{pushes}(up) + \sum_{cp \in \alpha} \mathit{pushes}(cp) \leq \sum_{up \in \alpha} [2 \cdot \mathit{attempts}(up) + h(up) + 4\dot{c}(up)] + \sum_{cp \in \alpha} [3 \cdot \mathit{attempts}(cp) + h(cp) + 10\dot{c}(cp) \cdot \mathit{rebal}(\mathit{viol}(cp))].$$

The majority of our amortized analysis is devoted to showing that

$$\sum_{up \in \alpha} \mathit{attempts}(up) \leq \sum_{up \in \alpha} [30h(up) + 5594\dot{c}(up) + 159]$$

and

$$\sum_{cp \in \alpha} \mathit{attempts}(cp) \leq \sum_{cp \in \alpha} [78h(cp) + 312\dot{c}(cp) + 5008\dot{c}(cp) \cdot \mathit{rebal}(\mathit{viol}(cp)) + 357].$$

Note that the constants used are larger than required to simplify the analysis. Combining these results with Lemma 9 gives the following lemma.

▶ **Lemma 10.** *For any finite execution α of* INSERT*,* DELETE*, and* FIND *operations on the chromatic tree,*

$$\sum_{op \in \alpha} \mathit{steps}(op) = O\Bigg(\sum_{up \in \alpha} [h(up) + \dot{c}(up)] + \sum_{cp \in \alpha} [h(cp) + \mathit{rebal}(\mathit{viol}(cp)) \cdot \dot{c}(cp)]\Bigg).$$

The height terms $h(up)$ and $h(cp)$ in Lemma 10 are the number of steps to perform a search in the update phase and cleanup phase, respectively, assuming there are no concurrent operations. The $\dot{c}(up)$ terms account for extra steps taken by operations concurrent with *up* due to the successful insertion or deletion made by *up*. Similarly, the $\mathit{rebal}(\mathit{viol}(cp)) \cdot \dot{c}(cp)$ terms account for extra steps taken due to successful rebalancing transformations centered at $\mathit{viol}(cp)$.

Lemma 10 and Lemma 3 together prove the main theorem of our paper.

▶ **Theorem 11.** *The amortized number of steps made by any chromatic tree operation op in any finite execution α is $O(\dot{c}(\alpha) + \log n(op))$, where $n(op)$ is the maximum number of nodes in the chromatic tree during op's execution interval.*

Proof. For any operation *op* with update phase *up* and cleanup phase *cp*, $h(up) \leq h(op)$ and $h(cp) \leq h(op)$ since the starting configurations of *up* and *cp* are contained in *op*. Therefore,

$$\sum_{up \in \alpha} h(up) + \sum_{cp \in \alpha} h(cp) \leq \sum_{op \in \alpha} 2h(op).$$

Similarly, $\dot{c}(up) \leq \dot{c}(\alpha)$ and $\dot{c}(cp) \leq \dot{c}(\alpha)$ since *up* and *cp* are intervals contained in α. Therefore, Lemma 10 can be expressed as

$$\sum_{op \in \alpha} \mathit{steps}(op) = O\Bigg(\sum_{op \in \alpha} [h(op) + \dot{c}(\alpha)] + \dot{c}(\alpha) \cdot \sum_{cp \in \alpha} \mathit{rebal}(\mathit{viol}(cp))\Bigg).$$

Suppose the execution α contains i Insert operations and d Delete operations. Then, by Lemma 3,

$$\begin{aligned}\dot{c}(\alpha) \cdot \sum_{cp \in \alpha} rebal(viol(cp)) &\leq \dot{c}(\alpha)(3i + d - 2) \\ &\leq \dot{c}(\alpha)(3i + 3d) \\ &\leq \sum_{op \in \alpha} 3\dot{c}(\alpha).\end{aligned}$$

So, the total number of steps taken by all operations in an execution α is

$$\sum_{op \in \alpha} steps(op) = O\bigg(\sum_{op \in \alpha} [\dot{c}(\alpha) + h(op)]\bigg).$$

It follows that the amortized cost of a single operation op is $O(\dot{c}(\alpha) + h(op)) = O(\dot{c}(\alpha) + \log n(op))$, by Lemma 4. ◀

Notice that the amortized cost of each operation has an additive term of $O(\dot{c}(\alpha))$, as opposed to $O(\dot{c}(op))$. This is because the operation with the largest contention in the execution may perform a lot of the rebalancing. Lemma 3 only gives an upper bound on the total number of rebalancing transformations that may occur in an execution, but does not state anything about which operations will perform these rebalancing transformations.

The analysis for bounding the total number of attempts made in update phases in the chromatic tree is similar to the analysis for bounding the total number of attempts made in the unbalanced binary search tree. In the remaining sections, we give an overview of the analysis for bounding the total number of attempts made in cleanup phases. The complete amortized analysis appears in the full version of the paper.

5.1 Bounding the Number of Failed Cleanup Attempts

In this section, we count the number of cleanup attempts that fail due to unsuccessful LLXs. Counting the number of cleanup attempts that fail due to unsuccessful SCXs is similar. Lemma 3 gives an upper bound on the number of successful cleanup attempts, so we focus on counting unsuccessful cleanup attempts.

Our analysis uses the accounting method. We define a number of bank accounts for each cleanup phase *cp*. The rules that deposit and withdraw dollars from bank accounts are designed so that the *bank* (i.e. the collection of all bank accounts) satisfies the following properties for any finite execution.

- Property P1: The total number of dollars deposited into the bank by an operation is $O(h(cp) + rebal(viol(cp)) \cdot \dot{c}(cp))$ during its cleanup phase *cp*.
- Property P2: Every failed attempt caused by a failed LLX during an operation's cleanup phase *cp* withdraws one dollar from one of its own accounts.
- Property P3: All bank accounts have non-negative balance.

Assuming these properties of the bank hold, the total number of attempts that fail due to LLXs is bounded by the total number of dollars deposited into the bank. Thus, the total number of attempts that fail due to LLXs is $\sum_{cp \in \alpha} O(h(cp) + rebal(viol(cp)) \cdot \dot{c}(cp))$.

Let $\#llx(cp)$ be the maximum number of LLXs performed by *cp* in a single attempt. It can be verified that $\#llx(cp) \leq 7$. For each node x in the chromatic tree at some point in the execution, and for each cleanup phase *cp*, we define the accounts $B(cp)$, $B_{llx}(cp, x)$, $S(cp, x)$ and $L_i(cp)$, for $1 \leq i \leq \#llx(cp)$. For each type of bank account $X(args)$, we let $X(args, C)$

Table 1 Rules for bank accounts owned by a cleanup phase *cp*.

D1-S	Consider a **successful update CAS** *ucas* for a rebalancing transformation centered at $viol(cp)$. For all nodes x removed from the chromatic tree by *ucas* and for all active cleanup phases cp', cp deposits 1 dollar into $S(cp', x)$.
D1-L	The **start of a cleanup phase** cp deposits 3 dollars into each of its own $L_i(cp)$ accounts (for $1 \leq i \leq \#llx(cp)$).
D2-L	A **successful update CAS** for a rebalancing transformation centered at $viol(cp)$ deposits 3 dollars into each active L_i account.
D1-B	The **start of a cleanup phase** cp deposits $78h(cp) + 312\dot{c}(cp) + 312$ dollars into its own $B(cp)$ account.
D2-B	A **successful update CAS** of a rebalancing transformation centered at $viol(cp)$ deposits 4934 dollars into each active B account.
D3-B	A **successful commit step** of a rebalancing transformation centered at $viol(cp)$ deposits 26 dollars into each active B account.
W-S	A **stale invocation** of TryRebalance for a cleanup phase cp that blames a node x withdraws 1 dollar from $S(cp, x)$.
W-L	Suppose that cp fails its ith LLX during a non-stale instance of TryRebalance. If $L_i(cp)$ is non-empty, the **failure step of the LLX** withdraws 1 dollar from $L_i(cp)$. If $L_i(cp)$ is empty, consider the node x on which this LLX is performed. If x is a downwards node for the SCX blamed by the LLX, then cp withdraws 1 dollar from $B_{llx}(cp, x)$; otherwise it withdraws from $B_{llx}(cp, p)$, where p is the parent of x.
T-B	Consider a **successful freezing CAS** from a configuration C performed by any process on a downwards node x for some SCX. Every cleanup phase cp, where $x \in targets(cp, C)$, transfers 1 dollar from $B(cp)$ to $B_{llx}(cp, x)$.

be the number of dollars in $X(args)$ in a configuration C. A bank account $X(cp)$ or $X(cp, x)$ is *active* if cp is active. Each bank account is initially empty.

The bank accounts $S(cp, x)$ and $L_i(cp)$ will serve two purposes. First, they pay for any failed attempts due to unsuccessful LLXs of cp due to steps that occur prior to the start of cp. Second, they will pay for cp's failed attempts due to unsuccessful LLXs of cp due to recent changes in the structure of the chromatic tree. Whenever the $S(cp, x)$ and $L_i(cp)$ accounts do not pay for a failed attempt, a dollar will be withdrawn from one of the $B_{llx}(cp, x)$ accounts instead. The $B(cp)$ account is only responsible for transferring dollars into the $B_{llx}(cp, x)$ accounts.

The rules for bank accounts owned by cp are summarized in Table 1. Some terms in the rules will be defined when introduced in the following sections. There are 6 deposit rules beginning with D. They indicate which accounts cp deposits dollars into. There are two withdraw rules beginning with W. They define when cp withdraws dollars from its accounts. There is one transfer rule T-B that transfers dollars from $B(cp)$ to $B_{llx}(cp, x)$.

We next show that Properties P1 and P2 are always satisfied. Recall that a successful SCX contains 1 successful update CAS, 1 successful commit step, and a number of successful freezing CASs.

▶ **Lemma 12.** *The total number of dollars deposited into the bank by an operation is* $O(h(cp) + rebal(viol(cp)) \cdot \dot{c}(cp))$ *during its cleanup phase cp.*

Proof. Consider a process in its cleanup phase cp. Rules D1-L and D1-B are each applied once, at the start of cp. Rule D1-L only deposits a constant number of dollars. Rule D1-B deposits $O(h(cp) + \dot{c}(cp))$ dollars. By definition, there are $rebal(viol(cp))$ successful

rebalancing transformations centered at $viol(cp)$, so each of rules D1-S, D2-L, D2-B, and D3-B are applied $rebal(viol(cp))$ times throughout cp. There are at most $\dot{c}(cp)$ active operations each time a rule is applied. It can be verified by inspection of the rebalancing transformations that at most 5 nodes are removed from the chromatic tree by a single update CAS, so rule D1-S deposits $O(\dot{c}(cp))$ dollars each time it is applied. Since $\#llx(cp) \leq 7$, rule D2-L deposits $O(\dot{c}(cp))$ dollars each time it is applied. By the rules in Table 1, cp deposits a total of $O(h(cp) + rebal(viol(cp)) \cdot \dot{c}(cp))$ dollars. Thus, Property P1 of the bank is satisfied. ◀

▶ **Lemma 13.** *Every failed attempt caused by a failed LLX during an operation's cleanup phase cp withdraws one dollar from one of its own accounts.*

Proof. Every cleanup attempt contains at most 1 unsuccessful LLX. The LLXs performed by cp only occur during an instance of TryRebalance. By rule W-S, if this instance is *stale* (which will be defined in the following section), a dollar is withdrawn from one of cp's S accounts. Otherwise, by rule W-L, a dollar is withdrawn from one of cp's L_i or B_{llx} accounts. Thus, Property P2 of the bank is satisfied. ◀

It remains to show that the balance of each bank account is non-negative. In Section 5.2, we consider the $S(cp, x)$, $L_i(cp)$, and $B_{llx}(cp, x)$ accounts, and in Section 5.3, we consider the $B(cp)$ accounts.

5.2 The $S(cp, x)$, $L_i(cp)$, and $B_{llx}(cp, x)$ Accounts

In this section, we describe the purpose of the $S(cp, x)$, $L_i(cp)$, and $B_{llx}(cp, x)$ accounts, and show that the balance in these accounts are non-negative.

▶ **Definition 14.** An instance I of TryRebalance(ggp, gp, p, v) starting from configuration C is called *stale* if one of ggp, gp, p, or v is not in the chromatic tree in C. If x is the first node in the sequence $\langle ggp, gp, p, v\rangle$ that is no longer in the chromatic tree in C, then I *blames* the node x.

The $S(cp, x)$ account pays for all failed attempts containing a stale instance of TryRebalance(ggp, gp, p, v) that blame a node x. According to D1-S, whenever cp is active and a node x is removed from the chromatic tree, 1 dollar is deposited into $S(cp, x)$. It can be shown that TryRebalance will blame a node x at most once sometime after x is removed from the chromatic tree, since in cp's following cleanup attempt, cp will restart its search from a node from which x is no longer reachable. It follows that $S(cp, x, C) \geq 0$.

The $L_i(cp)$ or $B_{llx}(cp, x)$ accounts are used to pay for each failed attempt due to an unsuccessful LLX during a non-stale instance of TryRebalance. We require a number of definitions to explain why at least one of these bank accounts has a positive balance when these LLXs occur. During a cleanup phase cp by a process P, P traverses from the root to a leaf, attempting to cleanup any violations it encounters during its traversal. When cp is performing a rebalancing transformation centered at a node v in the chromatic tree in a configuration C, we define $focalNode(cp, C) = v$. If not, but the node cp is currently visiting in its search is in the chromatic tree, then $focalNode(cp, C)$ is defined to be this node. Otherwise, $focalNode(cp, C)$ is the next node in the chromatic tree that cp will visit in a solo execution starting from C. Let $focalPath(cp, C)$ be the set of all nodes along the path from the root to $focalNode(cp, C)$, including both endpoints.

We define a set of nodes $rebalSet(v, C)$ so that any rebalancing transformation whose center node is v only involves nodes in $rebalSet(v, C)$. We use $rebalSet(v, C)$ to simplify

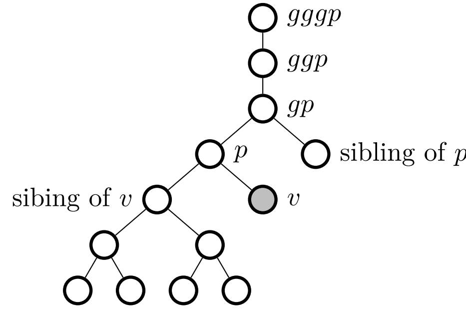


Figure 4 The set of nodes $rebalSet(v, C)$.

the amortized analysis so that each of the 11 rebalancing transformations do not have to be considered individually. This set of nodes is illustrated in Figure 4.

▶ **Definition 15.** For any node v in the chromatic tree in configuration C, let $rebalSet(v, C)$ be the set of nodes including the following 13 nodes:

- v and its four closest ancestors (labeled p, gp, ggp, and $gggp$),
- p's sibling, and
- all nodes with depth 2 or less in the subtree rooted at v's sibling.

For every cleanup phase cp and configuration C in which cp is active, let $targets(cp, C) = rebalSet(focalNode(cp, C), C)$. The following lemma concerning non-stale instances of TryRebalance is used to prove properties of the $L_i(cp)$ accounts.

▶ **Lemma 16.** *In the first configuration C of a non-stale* TryRebalance(ggp, gp, p, v) *performed by cp, $targets(cp, C) = rebalSet(v, C)$.*

For every unsuccessful LLX that occurs during a non-stale instance of TryRebalance, we choose an SCX which causes the LLX to fail.

▶ **Definition 17.** Let I be an LLX(r) that returns Fail or Finalized. Let the *failure step* of I be the last time I reads $r.info$, the pointer to r's SCX-record. Let U be the SCX-record pointed to by $r.info$ when I performs its failure step. We say that I *blames* the SCX that created U.

Consider a failed attempt by cp due to an unsuccessful LLX during a non-stale instance of TryRebalance. By rule W-L, if cp's ith LLX during this instance of TryRebalance is unsuccessful and $L_i(cp)$ is non-empty, then cp withdraws 1 dollar from $L_i(cp)$. Since this is the only rule that withdraws from $L_i(cp)$, the $L_i(cp)$ account always has a non-negative balance. Rules D1-L and D2-L show that $L_i(cp)$ is non-empty during cp's first 3 attempts, or when a successful update CAS (by some other process) has occurred during cp's previous 3 attempts. We can use these facts along with Lemma 16 to prove the following lemma.

▶ **Lemma 18.** *Suppose a cleanup phase cp fails its ith LLX during a non-stale instance of* TryRebalance. *Let x be the node on which this LLX is performed. If $L_i(cp)$ does not pay for the unsuccessful LLX(x), then $x \in targets(cp, C')$ in the configuration C' immediately before the successful freezing CAS on x during the SCX blamed by the LLX(x).*

If $L_i(cp)$ is empty, the failed attempt is paid for by a $B_{llx}(cp, x)$ account. No rules deposit directly into $B_{llx}(cp, x)$ accounts. Dollars are only transferred into $B_{llx}(cp, x)$ from $B(cp)$ by rule T-B. Lemma 18 can be used to show that each time rule W-L withdraws a dollar from $B_{llx}(cp, x)$, rule T-B has previously transferred a dollar into this account.

▶ **Lemma 19.** *Consider the failure step of an unsuccessful LLX(x) that withdraws 1 dollar from $B_{llx}(cp, x)$. In the configuration C immediately before the failure step, $B_{llx}(cp, x, C) > 0$.*

To see why Lemma 19 is true, let S be the SCX blamed by the unsuccessful LLX(x). Let C' be the configuration immediately before the successful freezing CAS on x during S. Since a dollar is withdrawn from $B_{llx}(cp, x)$ to pay for this LLX, Lemma 18 implies that $x \in targets(cp, C')$. Furthermore, by rule W-L, x is a downwards node for S. Therefore, by rule T-B, the successful freezing CAS that freezes x for S transfers 1 dollar from $B(cp)$ into $B_{llx}(cp, x)$. Using properties of LLX, we can prove that no other unsuccessful LLX by *cp* withdraws this dollar between C' and C. Therefore, $B_{llx}(cp, x, C) > 0$.

According to rule W-L, an unsuccessful LLX(x) may instead withdraw a dollar from $B_{llx}(cp, p)$, where p is the parent of x. Results similar to Lemma 18 and Lemma 19 show that, in this case, $B_{llx}(cp, p)$ is also non-empty. Therefore, the B_{llx} accounts always have non-negative balance.

5.3 The $B(cp)$ Accounts

Rule T-B is the only rule that reduces the number of dollars in $B(cp)$. Since unsuccessful SCXs may contain successful freezing CASs, the total number of dollars transferred from $B(cp)$ throughout *cp* may be large. We show that, whenever T-B is applied, the balance in $B(cp)$ remains non-negative. We will define a function $J(cp, C)$ and show that, for every configuration C, $B(cp, C) \geq J(cp, C) \geq 0$.

We require some definitions. For any variable $z(args, C)$, we use $z(args)$ to denote $z(args, C)$ as a function of C. For any node x in the chromatic tree in configuration C, let

- $depth(x, C)$ be the number of edges along the path from the root to x in C,
- $isFrozen(x, C)$ be 1 if x is frozen in C and 0 otherwise, and
- $npa(x, C)$ be the set of nodes that are not proper ancestors of x in C.

▶ **Definition 20.** For each node x in the chromatic tree at some point in an execution, define $abort(x, C)$, which is 0 in initial configurations and is updated by the following four rules:

- A1: A successful freezing CAS on a downwards node x sets $abort(p) = 1$, where p is the parent of x.
- A2: A successful freezing CAS on a cross node x sets $abort(x) = 1$.
- A3: A successful abort step that unfreezes node x sets $abort(x) = 0$.
- A4: The completion of an update or cleanup attempt sets $abort(x) = 0$ for all nodes x except for those on which LLX(x) has been performed in the latest attempt of an active operation.

Next we define $H(x, C)$, which maps node x in the chromatic tree in configuration C to an integer. A similar function is defined in the analysis of the unbalanced binary search tree [7]. For any node x in the chromatic tree in configuration C, let

$$H(x, C) = 3h(C) + 12 - 3depth(x, C) + 6\dot{c}(C) + \sum_{u \in npa(x,C)} [abort(u, C) - isFrozen(u, C)].$$

▶ **Lemma 21.** *For any node x in the chromatic tree in configuration C, $H(x, C)$ has the following properties.*

1. $0 \leq H(x, C) \leq 3h(C) + 12\dot{c}(C) + 12$.
2. *If p is the parent of x, $2 \leq H(p, C) - H(x, C) \leq 4$.*
3. *If s is the sibling of x, $H(s, C) = H(x, C)$.*

4. *A successful freezing CAS on a downwards node x of an SCX decreases $H(x)$ by 1, and does not increase $H(u)$ for any other node u in the chromatic tree.*
5. *A successful commit step of an SCX increases $H(x)$ by at most 1.*
6. *A successful abort step of an SCX does not increase $H(x)$.*

▶ **Definition 22.** For every cleanup phase cp and every node x in the chromatic tree at the beginning of cp, $backup(cp, x) = 0$. When an update CAS by any process adds a new node x into the chromatic tree during cp, $backup(cp, x)$ is initialized to 1 if x is on $focalPath(cp)$, and 0 otherwise. Finally, a step that removes a node x from $focalPath(cp)$ sets $backup(cp, x) = 0$.

If $backup(cp, x, C) = 1$, then x is a node that has been added as a proper ancestor of $focalNode(cp, C)$ since cp began. When cp backtracks up the tree, so that x is no longer on $focalPath(cp)$, the decrease in $backup(cp, x)$ will ensure that $J(cp)$ does not increase.

▶ **Definition 23.** For any cleanup phase cp and any configuration C in which cp is active,

$$J(cp, C) = \sum_{x \in targets(cp,C)} 2H(x, C) + \sum_{x \in focalPath(cp,C)} 576 \cdot backup(cp, x, C).$$

By Lemma 21.1, $H(x, C) \geq 0$ for all nodes x and configurations C. By definition, $backup(cp, x, C) \geq 0$. Hence $J(cp, C)$ is always non-negative.

▶ **Lemma 24.** *For all cleanup phases cp and configurations C in which cp is active, $B(cp, C) \geq J(cp, C) \geq 0$.*

Proof sketch. In the first configuration C' in which cp is active, $backup(cp, x, C') = 0$ for all nodes x. By Lemma 21.1, $2H(x, C') \leq 6h(C') + 24\dot{c}(C') + 24$. Since $|targets(cp, C')| = 13$, $J(cp, C') \leq 78h(C') + 312\dot{c}(C') + 312$. Thus, by rule D1-B, $B(cp, C') \geq J(cp, C')$. We show that, whenever a step subsequently decreases the balance in $B(cp)$, $J(cp)$ decreases by at least the same amount, and whenever a step subsequently increases $J(cp)$, the balance in $B(cp)$ increases by at least the same amount. This implies that, in all configurations C in which cp is active, $B(cp, C) \geq J(cp, C)$.

Rule T-B decreases the balance in $B(cp)$ by 1 when a successful freezing CAS is performed on a downwards node x in $targets(cp, C)$. Lemma 21.4 says that, when this happens, $H(x)$ decreases by 1 and the H values of all other nodes do not change, so $J(cp)$ decreases by 2. No other steps decrease the balance of $B(cp)$.

Only successful commit steps and update CASs increase $J(cp)$. For example, Lemma 21.2 and Lemma 21.3 can be used to show that, when $focalNode(cp)$ changes from a node p to one of its children x, $J(cp)$ decreases as a result of the changes to $targets(cp)$ and $focalPath(cp)$. Furthermore, by Lemma 21.6, abort steps do not increase $J(cp)$.

By Lemma 21.5, a commit step increases $H(x)$ by at most 1, for all nodes x in the chromatic tree. Thus, $J(cp)$ increases by at most $2|targets(cp, C)| = 26$. Rule D3-B deposits 26 dollars into $B(cp)$, and so $B(cp, C) \geq J(cp, C)$.

A successful update CAS can change $targets(cp)$. We use the fact that rebalancing transformations only change a localized section of the tree to calculate an upper bound on the difference between the H values of cp's new targets and old targets. This is done independently of the type of rebalancing transformation applied by the update CAS.

Because an update CAS may remove nodes from the chromatic tree that are on cp's stack, $focalNode(cp)$ may change to a new node with much smaller depth. As a result, each term in the first summation in $J(cp)$ may increase by an amount that is proportional to the change in depth of $focalNode(cp)$. The $backup(cp, x)$ variables are used to offset this increase. We

show that whenever $focalNode(cp)$ decreases in depth, a proportional number of nodes x with $backup(cp, x) = 1$ are removed from $focalPath(cp)$. This decreases the second summation in $J(cp)$ to offset the increase in the first summation in $J(cp)$. We show that $J(cp)$ increases by at most 4934 as a result of an update CAS, so the number of dollars deposited into $B(cp)$ by rule D2-B is sufficient to maintain that $B(cp, C) \geq J(cp, C)$. ◀

Since all bank accounts owned by cp have non-negative balance, Property P3 of the bank is satisfied. Thus, the total number of attempts that fail due to unsuccessful LLXs in an execution α is $\sum_{cp \in \alpha} O(h(cp) + rebal(viol(cp)) \cdot \dot{c}(cp))$.

6 Conclusion

We have shown that the amortized step complexity of an implementation of a non-blocking chromatic tree is $O(\dot{c}(\alpha) + \log n(op))$. It would be interesting to see if amortized step complexity $O(\dot{c}(op) + \log n(op))$ can be shown for the chromatic tree. This is more challenging because one must argue that an operation with high contention does not perform an excessive amount of rebalancing. This may require a more detailed analysis of the chromatic tree than what is shown by Lemma 3. Alternatively, one may try to show a lower bound of $\Omega(\dot{c}(\alpha) + \log n(op))$. Finally, we believe that the techniques presented here can be applied to other balanced binary search tree implementations using LLX and SCX.

References

1. Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. Wiley-Interscience, 2nd edition, 2004.
2. Joan Boyar, Rolf Fagerberg, and Kim S. Larsen. Amortization Results for Chromatic Search Trees, with an Application to Priority Queues. In *Proceedings of Algorithms and Data Structures, 4th International Workshop (WADS)*, pages 270–281, 1995. doi:10.1007/3-540-60220-8_69.
3. Trevor Brown. *Techniques for Constructing Efficient Lock-Free Data Structures*. PhD thesis, Department of Computer Science, University of Toronto, 2017.
4. Trevor Brown, Faith Ellen, and Eric Ruppert. Pragmatic primitives for non-blocking data structures. In *Proceedings of the Symposium on Principles of Distributed Computing (PODC)*, pages 13–22, 2013. doi:10.1145/2484239.2484273.
5. Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 329–342, 2014. doi:10.1145/2555243.2555267.
6. Bapi Chatterjee, Nhan Nguyen Dang, and Philippas Tsigas. Efficient lock-free binary search trees. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 322–331, 2014. doi:10.1145/2611462.2611500.
7. Faith Ellen, Panagiota Fatourou, Joanna Helga, and Eric Ruppert. The amortized complexity of non-blocking binary search trees. In *Proceedings of the Symposium on Principles of Distributed Computing (PODC)*, pages 332–340, 2014. doi:10.1145/2611462.2611486.
8. Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 131–140, 2010. doi:10.1145/1835698.1835736.
9. Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 50–59, 2004. doi:10.1145/1011767.1011776.

10 Joel Gibson and Vincent Gramoli. Why Non-blocking Operations Should be Selfish. In *Proceedings of the Distributed Computing - 29th International Symposium (DISC)*, pages 200–214, 2015. doi:10.1007/978-3-662-48653-5_14.

11 Timothy L. Harris. A Pragmatic Implementation of Non-blocking Linked-Lists. In *Proceedings of the Distributed Computing, 15th International Conference (DISC)*, pages 300–314, 2001. doi:10.1007/3-540-45414-4_21.

12 Otto Nurmi and Eljas Soisalon-Soininen. Chromatic Binary Search Trees. A Structure for Concurrent Rebalancing. *Acta Inf.*, 33(6):547–557, 1996. doi:10.1007/BF03036462.

13 Niloufar Shafiei. Non-Blocking Doubly-Linked Lists with Good Amortized Complexity. In *Proceedings of the 19th International Conference on Principles of Distributed Systems (OPODIS)*, pages 35:1–35:17, 2015. doi:10.4230/LIPIcs.OPODIS.2015.35.

Lock-Free Search Data Structures: Throughput Modeling with Poisson Processes

Aras Atalar
Chalmers University of Technology, S-41296 Göteborg, Sweden
aaras@chalmers.se

Paul Renaud-Goud
Informatics Research Institute of Toulouse, F-31062 Toulouse, France
Paul.Renaud.Goud@irit.fr

Philippas Tsigas
Chalmers University of Technology, S-41296 Göteborg, Sweden
philippas.tsigas@chalmers.se

Abstract

This paper considers the modeling and the analysis of the performance of lock-free concurrent search data structures. Our analysis considers such lock-free data structures that are utilized through a sequence of operations which are generated with a memoryless and stationary access pattern. Our main contribution is a new way of analyzing lock-free concurrent search data structures: our execution model matches with the behavior that we observe in practice and achieves good throughput predictions.

Search data structures are formed of basic blocks, usually referred to as nodes, which can be accessed by two kinds of events, characterized by their latencies; (i) CAS events originated as a result of modifications of the search data structure (ii) Read events that occur during traversals. An operation triggers a set of events, and the running time of an operation is computed as the sum of the latencies of these events. We identify the factors that impact the latency of such events on a multi-core shared memory system. The main challenge (though not the only one) is that the latency of each event mainly depends on the state of the caches at the time when it is triggered, and the state of caches is changing due to events that are triggered by the operations of any thread in the system. Accordingly, the latency of an event is determined by the ordering of the events on the timeline.

Search data structures are usually designed to accommodate a large number of nodes, which makes the occurrence of an event on a given node rare at any given time. In this context, we model the events on each node as Poisson processes from which we can extract the frequency and probabilistic ordering of events that are used to estimate the expected latency of an operation, and in turn the throughput. We have validated our analysis on several fundamental lock-free search data structures such as linked lists, hash tables, skip lists and binary trees.

2012 ACM Subject Classification Theory of computation → Concurrency

Keywords and phrases Lock-free, Search Data Structures, Performance, Modeling, Analysis

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.9

1 Introduction

A search data structure is a collection of $\langle key, value \rangle$ pairs which are stored in an organized way to allow efficient search, delete and insert operations. Linked lists, hash tables, binary trees are some widely known examples. Lock-free implementations of such concurrent data

© Aras Atalar, Paul Renaud-Goud, and Philippas Tsigas;
licensed under Creative Commons License CC-BY
22nd International Conference on Principles of Distributed Systems (OPODIS 2018).
Editors: Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira; Article No. 9; pp. 9:1–9:16

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

structures are known to be strongly competitive at tackling scalability by allowing processors to operate asynchronously on the data structure.

Performance (here throughput, *i.e.* number of operations per unit of time) is ruled by the number of events in a search data structure operation (*e.g.* $O(\log \mathcal{N})$ for the expected number of steps in a skip list or a binary tree). The practical performance estimation requires an additional layer as the cost (latency) of these events need to be mapped onto the hardware platform; typical values of latency varies from 4 cycles for an access to the first level of cache, to 350 cycles for the last level of remote cache. To estimate the latency of events, one needs to consider the misses, which are sensitive to the interleaving of these events on the timeline. On the one hand, a capacity miss in data or TLB (Translation Lookaside Buffer) caches with LRU (Least Recently Used) policy arise when the interleaving of memory accesses evicted a cacheline. On the other hand, the coherence cache misses arise due to the modifications, that are often realized with *Compare-and-Swap* (*CAS*) instructions, in the lock-free search data structure. The interleaving of events that originate from different threads, determine the frequency and severity of these misses, hence the latencies of the events.

In the literature, there exist many asymptotic analyses on the time complexity of sequential search data structures and amortized analyses for the concurrent lock-free variants that involve the interaction between multiple threads. But they only consider the number of events, ignoring the latency. On the other side, there are performance analyses that aim to estimate the coherence and capacity misses for the programs on a given platform, with no view on data structures. We go through them in the related work. However, there is a lack of results that merge these approaches in the context of lock-free data structures to analytically predict their practical performance.

An analytical performance prediction framework could be useful in many ways: (i) to facilitate design decisions by providing an extensive understanding; (ii) to compare different designs in various execution contexts; (iii) to help the tuning process. On this last point, lock-free data structures come with specific parameters, *e.g.* padding, back-off and memory management related parameters, and become competitive only after picking their hopefully optimal values.

In this paper, we aim to compute the average throughput ($\mathcal{T}$) of search data structures for a sequence of operations, generated by a memoryless and stationary access pattern. Throughput is directly linked to the latency of operations. As the traversal of a search data structure is light in computation, the latency of an operation is dominated by the memory access costs to the nodes that belong to the path from the entry of the data structure to the targeted node.

Therefore, part of this paper is dedicated to the discovery of the route(s) followed by a thread on its way to reach any node in the data structure. In other words, what is the sequence of nodes that are accessed when a given key is targeted by an operation.

As the latency of an operation is the sum of the latency of each memory access to the nodes that are on the path, we obviously need to estimate the individual latency of each accessed node. Even if, in the end, we are interested in the average throughput, this part of the analysis cannot be satisfied with a high-level approach, where we would ignore which thread accesses which node across time. For instance, the cache, whose misses are expected to greatly impact throughput, should be taken carefully into account. This can only be done in a framework from which the interleaving of memory accesses among threads can be extracted. That is why we model the distribution of the memory accesses for every thread.

More precisely, a memory access can be either the read or the modification of a node, and two point distributions per node represent the triggering instant of either a *Read* or a *CAS*.

These point distributions are modeled as Poisson processes, since they can be approximated by Bernoulli processes, in the context of rare events. Knowing the probabilistic ordering of these events gives a decisive information that is used in the estimate of the access latency associated with the triggered event. Once this information is grabbed, we roll back to the expectation of the access of a node, then to the expectation of the latency of an operation.

We validate our approach through a large set of experiments on several lock-free search data structures that are based on various algorithmic designs, namely linked lists, hash tables, skip lists and binary trees. We feed our experiments with different key distributions, and show that our framework is able to predict and explain the observed phenomena.

The rest of the paper is organized as follows. We discuss related work in Section 2, then the problem is formulated in Section 3. We present our framework in Section 4 and analysis of throughput in Section 5. In Section 6, we show how to initiate our model by considering the particularity of different search data structures. Finally, we describe the experimental results in Sections 7.

2 Related Work

When it comes to estimating the performance of concurrent lock-free data structures, the focus of the previous work has been on studying the contention overhead that occurs due to the existence of concurrent operations which overlap in time and access the same shared memory locations. In such cases, the contention manifests in the form of stall times [13] due to hardware conflicts and extended operation execution times due to logical conflicts in the algorithmic level (*e.g.* re-execution of the retry loop iteration).

For the time complexity of lock-free search data structure operations, previous work considered asymptotic amortized analysis [16] since it is not possible to bound the execution time of a single operation, by definition. The analysis is parameterized with a measure of contention (*e.g.* point contention [5]) that bounds the extra cost of failed attempts that are billed to a successful operation. Also, it is common to model a concurrent execution as an adversary that schedules the steps of concurrent processes in order to analyze the impact of contention [8, 13]. These studies target theoretical worst-case execution times. Closer to the practical domain, the *expected* system and individual operation latencies are analyzed for a general class of lock-free algorithms under a uniform stochastic scheduler in [1].

In our previous work [2, 3], we aim at estimating the average throughput performance of some lock-free data structures, that is observed in practice. We consider a universal construction [20] of lock-free data structures in its practical use. Data structures that have inherent sequential bottlenecks (*e.g.* stacks, counters and queues) are targeted; the universal construction can not be exploited in practice for the *efficient* designs of search data structures since it inhibits the potential disjoint access parallelism.

Conversely, in this work, we study the performance of the efficient designs of lock-free search data structures. These designs employ fine-grained synchronization, in which the modifications are spread to many different shared memory locations. As a result of this characteristic, hardware (stall time) and logical conflicts between threads, that were playing a central role in our previous work, occurs very rarely for search data structures and the performance is driven by different impacting factors. The performance of concurrent lock-free search data structures is studied and investigated through empirical studies in [18, 11]. To the best of our knowledge, we attempt for the first time to model and analyze the performance of lock-free search data structures and obtain estimates that are close to what is observed in practice on top of actual hardware platforms.

Procedure AbstractAlgorithm

```
1 while ! done do
2     key ← SelectKey(keyPMF);
3     operation ← SelectOperation(operationPMF);
4     result ← SearchDataStructure(key, operation);
```

Figure 1 Generic framework.

On the other hand, various performance metrics for search data structures have been studied for the sequential setting. The search path length of skip lists is analyzed in [21]. In [12, 24], various performance shapers for the randomized trees are studied, such as the time complexity of operations, the expectation, and distribution of the depth of the nodes based on their keys. However, these studies are not concerned with the interaction between the algorithms and the hardware. The following approaches rely on the independent reference model (IRM) for memory references and derive theoretical results or performance analysis. In [15], the exact cache miss ratio is derived analytically (computationally expensive) for LRU caches under IRM. As an outcome of this approach, the cache miss ratio of a static binary tree is estimated by assigning independent reference probabilities to the nodes in [14].

3 Problem Statement

We describe in this section the structure of the algorithm and the system that is covered by our model. We target a multicore platform where the communication between threads takes place through shared memory accesses. The threads are pinned to separate cores and call AbstractAlgorithm (see Figure 1) when they are spawned.

A concurrent search data structure is a shared collection of data elements, each associated with a key, that support three basic operations holding a key as a parameter. `Search` (resp. `Insert`, `Delete`) operation returns (resp. inserts, deletes) the element if the associated key is present (resp. absent, present) in the search data structure, otherwise returns *null*.

The applications that use a search data structure can be seen as a sequence of operations on the structure, interleaved by application-specific code containing at least the key and operation selection, as reflected in AbstractAlgorithm.

The access pattern (*i.e.* the output of the key and operation selections) should be considered with care since it plays a decisive role in the throughput value. An application that always looks for the first element of a linked list will obviously lead to very high throughput rates. In this study, we consider a memoryless and stationary key and operation selection process *i.e.* such that for any operation in the sequence the probability of selecting a key (resp. an operation type) is a constant.

A search data structure is modeled as a set of basic blocks called nodes, which either contain a value (*valued nodes*) or routes towards nodes (*router nodes*). W.l.o.g. the key set can be reduced to $[1..\mathcal{R}]$, where $\mathcal{R}$ is the number of possible keys. We denote by $(N_i)_{i\in[1..\mathcal{N}]}$ the set of $\mathcal{N}$ potential nodes, and by K_i the key associated with N_i. Until further notice , we assume that we have exactly one node per cacheline.

An operation can trigger two types of events in a node. We distinguish these events as *Read* and *CAS* events. The latency of an event is based on the state of the hardware platform at the time that the event occurs, *e.g.* for a *Read* request, the level of the cache that a node belongs to. We summarize the parameters of our model as follows:

- *Algorithm parameters:* Expected latency of the application specific-code (interleaves data structure operations) t^{app}, local computational cost while accessing a node t^{cmp} (a constant cost of a few cycles for the key comparisons, local updates for pointer chasing), probability mass functions for the key and operation selection.
- *Platform parameters:* Cache hit latencies (resp. capacity) from level ℓ: t_ℓ^{dat} (resp. C_ℓ^{dat}) for the data caches and t_ℓ^{tlb} (resp. C_ℓ^{tlb}) for TLB caches; other memory instruction latencies (that depends on P): t^{cas} for a *CAS* execution and t^{rec} to recover from an invalid state (*Read* at an invalid cacheline, that is in Modified state in another thread's local cache); number of threads P.

4 Framework

4.1 Event Distributions

We consider first a single thread running AbstractAlgorithm on a data structure where only search operations happen, and we observe the distribution of the *Read* triggering events on a given node N_i. The execution is composed of a sequence of search operations, where each operation is associated with a set of accessed nodes, which potentially includes N_i. If we slice the time into consecutive intervals, where an interval begins with a call to an operation, we can model the *Read* events as a Bernoulli process (a successful Bernoulli trial implies that a *Read* event on N_i occurs in the respective operation), where the probability of having a *Read* event during an interval depends on the parameters of the associated operation (recall that the process that generates the operation parameters is stationary and memoryless).

Search data structures have been designed as a way to store large data sets while still being able to reach any node within a short time: the set of accessed nodes is then expected to be small compared to the total number of nodes. This implies that, given an operation, the probability that N_i belongs to the set of accessed nodes is small. Therefore we can map the Bernoulli process on the timeline with constant-sized interval of length $\mathcal{T}^{-1}$ instead of mapping it with the actual operation intervals: as the probability of having a *Read* event within an operation is small, the duration between two events is big, and this duration is close to the number of initial intervals within this duration, multiplied by $\mathcal{T}^{-1}$ (with high probability, because of the Central Limit Theorem).

When we increase the scope of the operations to insertion and deletion, the structure is no longer static and the probability for a node to appear in an interval is no longer uniform, since it can move inside the data structure. There exists a long line of research in approximating Bernoulli processes by Poisson point processes [7], in which not only the number events but also their respective locations on the timeline are approximated. In particular, [9] has dealt with non-uniform Bernoulli processes, in which the success probabilities of trials are not necessarily same. Their error bounds, which are proportional to the success probabilities, strengthen the use of Poisson processes in our context: the events on N_i are rare, thus the probabilities in Bernoulli trials are small and the approximation is well-conditioned.

Once *Read* and *CAS* triggering events are modeled as Poisson processes for a single thread, the merge (superposition) of several Poisson processes models the multi-thread execution.

Lastly, we specify a point on the dynamicity: since we have insertions and deletions, nodes can enter and leave the data structure. This is modeled by the masking random variable P_i which expresses the presence of N_i in the structure. At a random time, we denote by D the set of nodes that are inside the data structure, and P_i is set to 1 iff $N_i \in D$. We denote by p_i its probability of success ($p_i = \mathbb{P}[P_i = 1]$). Its evaluation will often rely on the probability that the last update operation on key k was an `Insert`; we denote it by q_k, and

$q_k = \mathbb{P}\left[Op = op_k^{ins}\right] / (\mathbb{P}\left[Op = op_k^{ins}\right] + \mathbb{P}\left[Op = op_k^{del}\right])$. Note that the search data structures contain generally several *sentinel nodes* which define the boundaries of the structure and are never removed from the structure: their presence probability is 1.

For a given node N_i, we denote by λ_i^{acc} (resp. λ_i^{read}, λ_i^{cas}) the rate of the events triggering an access (resp. *Read*, *CAS*) of N_i due to one thread, when $N_i \in D$. op_k^{del} (resp. op_k^{ins}, op_k^{src}) stands for a `Delete` (resp. `Insert`, `Search`) on node key k. The probability for the application to select op_k^o, where $o \in \{ins, del, src\}$ is denoted by $\mathbb{P}\left[Op = op_k^o\right]$. $op_k^o \rightsquigarrow cas(N_i)$ (resp. $read\ (N_i)$) means that during the execution of op_k^o, a *CAS* (resp. a *Read*) occurs on N_i. Putting all together, we derive the rate of the triggering events:

$$\forall e \in \{cas, read\} \ : \ \lambda_i^e = \frac{\mathcal{T}}{P} \times \sum_{o \in \{ins,del,src\}} \sum_{k=1}^{\mathcal{R}} \mathbb{P}\left[Op = op_k^o\right] \times \mathbb{P}\left[op_k^o \rightsquigarrow e(N_i) \mid N_i \in D\right] \quad (1)$$

Recall for later that Poisson processes have useful properties, *e.g.* merging two Poisson processes produces another Poisson process whose rate is the sum of the two initial rates. This implies especially that the access triggering events follows a Poisson process with rate $\lambda_i^{acc} = \lambda_i^{read} + \lambda_i^{cas}$, and that the read triggering events that originates from P' different threads and occurs at N_i follow a Poisson process with rate $P' \times \lambda_i^{read}$.

To quantify the error in Poisson process approximations, we experimentally extract the cumulative distribution function of the inter-arrival latency of events that occur on a given node. Then, we apply the Kolmogorov-Smirnov test to compare it against exponential distributions (recall that the time between events in a Poisson process is exponentially distributed). Please see [4] for this comprehensive set of comparisons.

4.2 Impacting Factors

We have identified five factors that dominate the access latency of a node, distributed into two sets. On the one hand, the first set of factors only emerges in the concurrent executions as a result of the coherence issues on the search data structures. Atomic primitives, such as a *CAS*, are used to modify the shared search data structures asynchronously. To execute a *CAS* in multi-core architectures, the cache coherency protocol enforces exclusive ownership of the target cacheline by a thread (pinned to a core) through the invalidation of all the other copies of the cacheline in the system, if needed. One can guess the performance implications of this process that triggers back and forth communication among the cores. As the first factor, *CAS* instruction has a significant latency. The thread that executes the *CAS* pays this latency cost. Secondly, any other thread has to stall until the end of the *CAS* execution if it attempts to access (read or modify) the node while the *CAS* is getting executed. Last and most importantly, any thread pays a cost to bring a cacheline to a valid state if it attempts to access a node that resides in this cacheline and that has been modified by another thread after its previous access to this node.

On the other hand, the capacity misses in the data and TLB caches are other performance impacting factors for the node accesses. Consider a cache of size C, assume a node is accessed by a thread at time t and the next access (same thread and node) occurs at time t'. The thread would experience a capacity miss for the access at time t' if it has accessed at least C distinct nodes in the interval (t, t'). The same applies for TLB caches where the references to the distinct pages are counted instead of the nodes.

At a given instant, we denote by $Access_i$ the latency of accessing node N_i, either due to a *Read* event or a *CAS* event, for a given thread. This latency is the sum of random variables that correspond to the previous respective five impacting factors and the constant

local computation cost (≈ 4 cycles):

$$Access_i = t^{cmp} + CAS_i^{exe} + CAS_i^{stall} + CAS_i^{reco} + \sum_{\ell} Hit_i^{cache_\ell} + \sum_{\ell} Hit_i^{tlb_\ell}, \tag{2}$$

where, at a random time, CAS_i^{exe} is the latency of a CAS, CAS_i^{stall} the stall time implied by other threads executing a CAS on N_i, CAS_i^{reco} the time needed to fetch the data from another modifying thread, $Hit_i^{cache_\ell}$ the latency resulting from a hit on the data cache in level ℓ, and $Hit_i^{tlb_\ell}$ the latency coming from a hit on the TLB cache in level ℓ.

4.3 Solving Process

The solving decomposes into three main steps. Firstly, we can notice that Equation 1 exposes $2\mathcal{R}+1$ unknowns (the $2\mathcal{R}$ access rates and throughput) against $2\mathcal{R}$ equations. To end up with a unique solution, a last equation is necessary. The first two steps provide a last sufficient equation thanks to Little's law (see Section 5.2), which links throughput with the expectation of the access latency of a node, computed in Sections 5.1. We show in this section that the each component of the access latency can be expressed according to the access rates λ_i^{read} and λ_i^{cas}. The last step focuses on the values of the probabilities in Equation 1, which are strongly related with the particular data structure under consideration; they are instantiated in Section 6.1 (resp. 6.2, 6.3, 6.4) for linked lists (resp. hash tables, skip lists, binary trees).

5 Throughput Estimation

5.1 Access Latency

Applying expectation to Equation 2 leads to $\mathbb{E}\left[Access_i\right] = t^{cmp} + \mathbb{E}\left[CAS_i^{exe}\right] + \mathbb{E}\left[CAS_i^{stall}\right] + \mathbb{E}\left[CAS_i^{reco}\right] + \mathbb{E}\left[\sum_\ell Hit_i^{cache_\ell}\right] + \mathbb{E}\left[\sum_\ell Hit_i^{tlb_\ell}\right]$. We express here each term according to the rates at every node $\lambda_\star^{cas}$ and $\lambda_\star^{read}$.

CAS Execution

Naturally, among all access events, only the events originating from a CAS event contribute, with the latency t^{cas} of a CAS: $\mathbb{E}\left[CAS_i^{exe}\right] = t^{cas} \cdot \lambda_i^{cas}/(\lambda_i^{read} + \lambda_i^{cas})$.

Stall Time

A thread experiences stall time while accessing N_i when a thread, among the $(P-1)$ remaining threads, is currently executing a CAS on the same node. As a first approximation, supported by the rareness of the events, we assume that at most one thread will wait for the access to the node.

Firstly, we obtain the rate of CAS events generated by $(P-1)$ threads through the merge of their Poisson processes. Consider an access of N_i at a random time; (i) the probability of being stalled is the ratio of time when N_i is occupied by a CAS of $(P-1)$ threads, given by: $\lambda_i^{cas}(P-1)t^{cas}$; (ii) the stall time that the thread would experience is distributed uniformly in the interval $[0, t^{cas}]$. Then, we obtain: $\mathbb{E}\left[CAS_i^{stall}\right] = \lambda_i^{cas}(P-1)t^{cas}(t^{cas}/2)$.

Invalidation Recovery

Given a thread, a coherence cache miss occurs if N_i is modified by any other thread in between two consecutive accesses of N_i. The events that are concerned are: (i) the *CAS* events from any thread; (ii) the *Read* events from the given thread. When N_i is accessed, we look back at these events, and if among them, the last event was a *CAS* from another thread, a coherence miss occur: $\mathbb{P}[\text{Coherence Miss on } N_i] = \frac{\lambda_i^{cas}(P-1)}{\lambda_i^{cas}P+\lambda_i^{read}}$. We derive the expected latency of this factor during an access at N_i by multiplying this with the latency penalty of a coherence cache miss: $\mathbb{E}[CAS_i^{reco}] = \mathbb{P}[\text{Coherence Miss on } N_i] \times t^{rec}$.

Che's Approximation

Che's Approximation [10] is a technique to estimate the hit ratio of an LRU cache of size C, where the object (here, node) accesses follow IRM (Independent Reference Model). IRM is based on the assumption that the object references occur in an infinite sequence from a fixed catalog of $\mathcal{N}$ objects. The popularity of object i (denoted by s_i, where $i \in [1..\mathcal{N}]$) is a constant that does not depend on the reference history and does not vary over time.

Starting from $t = 0$, let the time of reference to object i be denoted by O_i, then the time for C unique references is given by: $t_c = \inf\{t > 0 : X(t) = C\}$, where $X(t) = \sum_{j=1}^{\mathcal{N}} \mathbf{1}_{0<O_j\leq t}$. Che's approximation estimates the hit ratio of object i with $Hit_i \approx 1 - e^{-s_i T}$, where the so-called characteristic time (denoted by T that approximates t_c) is the unique solution of the following equation: $C = \sum_{j=1}^{\mathcal{N}}(1 - e^{-s_j T}) = \mathbb{E}[X(T)]$. The accuracy of the approximation is rooted at the random variable $X(t)$ that is approximately Gaussian since it is defined as the sum of many independent random variables (Central Limit Theorem). We provide a more detailed discussion on this approximation in [4] based on the analysis in [17].

Cache Misses

We consider a data cache at level ℓ of size C_ℓ^{dat} and compute the hit ratio of N_i on this cache. On a given data structure, we have compared two related scenarios: a search only scenario, leading to a given expected size of the data structure, against a scenario with updates, that leads to the same expected size. We have observed that the capacity cache miss ratio were similar in both cases. Therefore, we consider that with or without updates, $N_\star$ is either present in the search data structure or not, *during the characteristic time of the cache.*

We can employ the access rates as popularities, *i.e.* $s_x = \lambda_x^{acc}$ for $x \in [1..\mathcal{N}]$, and modify Che's approximation to distinguish whether, at a random time, N_x is inside the data structure or not.

We integrate the masking variable P_x into Che's approximation. We have: $X^{cache}(t) = \sum_{x=1}^{\mathcal{N}} P_x \mathbf{1}_{0<O_x\leq t}$, where O_x denotes the reference time of N_x. We can still assume $X^{cache}(t)$ is Gaussian, as a sum of many independent random variables. We estimate the characteristic time as follows with the linearity of expectation and the independence of the random variables: $\mathbb{E}\left[X^{cache}(t)\right] = \sum_{x=1}^{\mathcal{N}} \mathbb{E}[P_x \mathbf{1}_{0<O_x\leq t}] = \sum_{x=1}^{\mathcal{N}} \mathbb{E}[P_x]\,\mathbb{E}[\mathbf{1}_{0<O_x\leq t}] = \sum_{x=1}^{\mathcal{N}} p_x(1 - e^{-\lambda_x^{acc} t})$. Lastly, we solve the equation for the characteristic time T_ℓ^{dat} of level ℓ cache: $\sum_{x=1}^{\mathcal{N}} p_x(1 - e^{-\lambda_x^{acc} T_\ell^{dat}}) = C_\ell^{dat}$ thanks to a fixed-point approach. After computing T_ℓ^{dat}, we estimate the cache hit ratio (on level ℓ) of N_i: $1 - e^{-\lambda_i^{acc} T_\ell^{dat}}$.

Page Misses

In this paragraph, we aim at computing the page hit ratio of N_i for the TLB cache at level ℓ of size C_ℓ^{tlb}. The total number $\mathcal{M}$ of pages that are used by the search data structure can be regulated by a parameter of the memory management scheme (frequency of recycling attempts

for the deleted nodes), as the total number of nodes is a function of $\mathcal{R}$. Different from the cachelines (corresponding to the nodes), we can safely assume that a page accommodates at least a single node that is present in the structure at any time.

We cannot apply straightforwardly Che's approximation since the page reference probabilities are unknown. However, we are given the cacheline reference probabilities $s_x = \lambda_x^{acc}$ for $x \in [1..\mathcal{N}]$ and we assume that $\mathcal{N}$ cachelines are mapped uniformly to $\mathcal{M}$ pages, $[1..\mathcal{N}] \rightarrow [1..\mathcal{M}]$, $\mathcal{N} > \mathcal{M}$. Under these assumptions, we know that the resulting page references would follow IRM because aggregated Poisson processes form again a Poisson process.

We follow the same line of reasoning as in the cache miss estimation. First, we consider a set of Bernoulli random variables (Y_x^j), leading to a success if N_x is mapped into page j, with probability $p_x/\mathcal{M}$ (hence Y_x^j does not depend on j). Under IRM, we can then express the page references as point processes with rate $r_j = \sum_{x=1}^{\mathcal{N}} Y_x^j s_x$, for all $j \in [1..\mathcal{M}]$.

Similar to the previous section, we denote the time of a reference to page j with O_j and we define the random variable $X^{page}(t) = \sum_{j=1}^{\mathcal{M}} \mathbf{1}_{0<O_j\leq t}$ and compute its expectation:

$$
\begin{aligned}
\mathbb{E}\left[X^{page}(t)\right] &= \sum_{j=1}^{\mathcal{M}} \mathbb{E}\left[\mathbf{1}_{0<O_j\leq t}\right] = \sum_{j=1}^{\mathcal{M}} \mathbb{E}\left[1 - e^{-r_j t}\right] = \sum_{j=1}^{\mathcal{M}} \mathbb{E}\left[1 - e^{-\sum_{x=1}^{\mathcal{N}} Y_x^j \lambda_x^{acc} t}\right] \\
&= \sum_{j=1}^{\mathcal{M}} \left(1 - \prod_{x=1}^{\mathcal{N}} \mathbb{E}\left[e^{-Y_x^j \lambda_x^{acc} t}\right]\right) = \sum_{j=1}^{\mathcal{M}} \left(1 - \prod_{x=1}^{\mathcal{N}} \left(\frac{\mathcal{M} - p_x}{\mathcal{M}} + \frac{p_x e^{-\lambda_x^{acc} t}}{\mathcal{M}}\right)\right) \\
\mathbb{E}\left[X^{page}(t)\right] &= \mathcal{M}\left(1 - \prod_{x=1}^{\mathcal{N}} \left(\frac{\mathcal{M} - p_x}{\mathcal{M}} + \frac{p_x e^{-\lambda_x^{acc} t}}{\mathcal{M}}\right)\right),
\end{aligned}
$$

Assuming $X^{page}(t)$ is Gaussian as it is sum of many independent random variables, we solve the following equation for the constant T_ℓ^{tlb} (characteristic time of a TLB cache of size C): $\mathbb{E}\left[X^{page}(T_\ell^{tlb})\right] = C_\ell^{tlb}$.

Lastly, we obtain the TLB hit rate for N_i by relying on the average *Read* rate of the page that N_i belongs to; we should add to the contributions of N_i, the references to the nodes that belong to the same page as N_i. Then follows the TLB hit ratio: $1 - e^{-z_i T_\ell^{tlb}}$, where $z_i = \lambda_i^{acc} + \mathbb{E}\left[\sum_{x=1, x\neq i}^{\mathcal{N}} Y_x^j \lambda_x^{acc}\right] = \lambda_i^{acc} + \sum_{x=1,x\neq i}^{\mathcal{N}} p_x \lambda_x^{acc}/\mathcal{M}$.

Interactions

To be complete, we mention the interaction between impacting factors and the possibility of latency overlaps in the pipeline. Firstly, the access latency of different nodes can not be overlapped due to the semantic dependency for the linked nodes. For a single node access, the latency for *CAS* execution and stall time can not be overlapped with any other factor. Based on the cache coherency protocol behavior, we do not charge invalidation recovery cost for *CAS* events. We consider inclusive data and TLB caches. It is not possible to have a cache hit on level l, if the cache on level $l-1$ is hit, and we do not consider any cost for the data cache hit if invalidation recovery or *CAS* execution (coherence) cost is induced (*i.e.* $\mathbb{E}\left[Hit_i^{cache_\ell}\right] = (1 - \mathbb{P}\left[coherence\ cost\right])(\mathbb{P}\left[hit\ cache_l\right] - \mathbb{P}\left[hit\ cache_{l-1}\right])t_\ell^{dat}$).

5.2 Latency vs. Throughput

In the previous sections, we have shown how to compute the expected access latency for a given node. There remains to combine these access latencies in order to obtain the throughput of the search data structure. Given $N_i \in D$, the average arrival rate of threads to N_i is

$\lambda_i^{acc} = \lambda_i^{read} + \lambda_i^{cas}$. Thus the average arrival rate of threads to N_i is: $p_i\lambda_i^{acc}$. It can then be passed to Little's Law [22], which states that the expected number of threads (denoted by t_i) accessing N_i obeys to $t_i = p_i\lambda_i^{acc}\mathbb{E}[Access_i]$. The equation holds for any node in the search data structure, and for the application call occurring in between search data structure operations. Its expected latency is a parameter ($\mathbb{E}[Access_0] = t^{app}$) and its average arrival rate is equal to the throughput ($\lambda_0^{acc} = \mathcal{T}$). Then, we have: $\sum_{i=0}^{\mathcal{N}} t_i = \sum_{i=0}^{\mathcal{N}}(p_i\lambda_i^{acc}\mathbb{E}[Access_i])$, where λ_i^{acc} and $\mathbb{E}[Access_i]$ are linear functions of $\mathcal{T}$. We also know $\sum_{i=0}^{\mathcal{N}} t_i = P$ as the threads should be executing some component of the program. We define constants with a_i, b_i, c_i for $i \in [0..\mathcal{N}]$. And, we represent $\lambda_i^{acc} = a_i\mathcal{T}$ and $\mathbb{E}[Access_i] = b_i\mathcal{T} + c_i$ and we obtain the following second order equation: $\sum_{i=0}^{\mathcal{N}}(p_i a_i b_i)\mathcal{T}^2 + \sum_{i=0}^{\mathcal{N}}(p_i a_i c_i)\mathcal{T} - P = 0$. This second order equation has a unique positive solution that provides the throughput, $\mathcal{T}$.

6 Instantiating the Throughput Model

In this section, we show how to instantiate our model with widely known lock-free search data structures, that have different operation time complexities. In order to obtain a throughput estimate for a structure, we need to compute the rates $\lambda_\star^{read}$ and $\lambda_\star^{cas}$, and $\mathbb{P}[op_k^o \rightsquigarrow e(N_i) \,|N_i \in D]$, *i.e.* the probability that, at a random time, an operation of type o on key k leads to a memory instruction of type e on node N_i, knowing that N_i is in the data structure. For the ease of notation, nodes will sometimes be doubly or triply indexed, and when the context is clear, we will omit $|N_i \in D$ in the probabilities. As a remark, the properties of the access pattern (memoryless and stationary) are critical here because they allow us to extract the probability of the state that the data structure could be in (based on the presence of nodes) at a random time, which is then used to find $\mathbb{P}[op_k^o \rightsquigarrow e(N_i) \,|N_i \in D]$.

We first estimate the throughput of linked lists and hash tables, on which we can directly apply our method, then we move on more involved search data structure, namely skip lists and binary trees, that need a particular attention.

6.1 Linked List

We start with the lock-free linked list implementation of Harris [19]. All operations in the linked list start with the search phase in which the linked list is traversed until a key. At this point all operations terminate except the successful update operations that proceed by modifying a subset of nodes in the structure with *CAS* instructions. The structure contains only valued node and two sentinel nodes N_0 and $N_{\mathcal{R}+1}$, so that $\mathcal{N} = \mathcal{R} + 2$ and for all $i \in [1..\mathcal{R}]$, N_i holds key i, *i.e.* $K_i = i$.

First, we need to compute the probabilities of triggering a *Read* event and *CAS* event on a node, given that the node is in the search data structure, for all operations of type $t \in \{\texttt{Insert}, \texttt{Delete}, \texttt{Search}\}$ targeted to key k.

At a random time, N_k, for $k \in [1..\mathcal{R}]$, is in the linked list iff the last update operation on key k is an insert: $p_k = q_k$, by definition of q_k. Moreover, when N_k is in the structure (condition that we omit in the notation), $op_{k'}^t$ reads N_k, either if N_k is before $N_{k'}$, or if it is just after $N_{k'}$. Formally, $\mathbb{P}[op_{k'}^o \rightsquigarrow read(N_k)] = 1$ if $k \leq k'$ and $\mathbb{P}[op_{k'}^o \rightsquigarrow read(N_k)] = \prod_{i=k'}^{k-1}(1-p_i)$ if $k > k'$.

CAS events can only be triggered by successful `Insert` and `Delete` operations. A successful `Insert` operation, targeted to $N_{k'}$, is realized with a *CAS* that is executed on N_k, where $k = \sup\{\ell < k' : N_\ell \in D\}$. The probability of success, which conditions the *CAS*'s, follows from the presence probabilities:

$$\mathbb{P}\left[op_{k'}^{ins} \rightsquigarrow cas(N_k)\right] = \begin{cases} 0, & \text{if } k \geq k' \\ \prod_{i=k+1}^{k'} (1-p_i), & \text{if } k < k' \end{cases} ;$$

$$\mathbb{P}\left[op_{k'}^{del} \rightsquigarrow cas(N_k)\right] = \begin{cases} 1, & \text{if } k = k' \\ 0, & \text{if } k > k' \\ p_{k'} \prod_{i=k+1}^{k'-1} (1-p_i), & \text{if } k < k' \end{cases}$$

6.2 Hash Table

We analyze here a chaining based hash table where elements are hashed to B buckets implemented with the lock-free linked list of Harris [19]. The structure is parametrized with a load factor lf which determines B through $B = \mathcal{R}/lf$. The hash function $h : k \mapsto \lceil k/lf \rceil$ maps the keys sequentially to the buckets, so that, after including the sentinel nodes (2 per bucket), we can doubly index the nodes: $N_{b,k}$ is the node in bucket b with key k, where $b \in [1..B]$ and $k \in [1..lf]$ (the last bucket may contain less elements).

$$\mathbb{P}\left[op_{b',k'}^{o} \rightsquigarrow read\big(N_{b,k}\big)\right] = \begin{cases} 0, & \text{if } b' \neq b \\ 1, & \text{if } b' = b \text{ and } k' \geq k \\ \prod_{j=k'}^{k-1} (1-p_{b,j}), & \text{if } b' = b \text{ and } k' < k \end{cases}$$

$$\mathbb{P}\left[op_{b',k'}^{ins} \rightsquigarrow cas\big(N_{b,k}\big)\right] = \begin{cases} 0, & \text{if } b' \neq b \text{ or } k' \leq k \\ \prod_{j=k+1}^{k'} (1-p_{b,j}), & \text{if } b' = b \text{ and } k' > k \end{cases}$$

$$\mathbb{P}\left[op_{b',k'}^{del} \rightsquigarrow cas\big(N_{b,k}\big)\right] = \begin{cases} 0, & \text{if } b' \neq b \text{ or } k' < k \\ 1, & \text{if } b' = b \text{ and } k' = k \\ p_{b,k'} \prod_{j=k+1}^{k'-1} (1-p_{b,j}), & \text{if } b' = b \text{ and } k' > k \end{cases}$$

In the previous two data structures, we do observe differences in the access rate from node to node, but the node associated with a given key does not show significant variation in its access rate during the course of the execution: inside the structure, the number of nodes preceding (and following) this node is indeed rather stable. In the next two data structures, node access rates can change dramatically according to node characteristics, that may include its position in the structure. In a skip list, a node N_i containing key K_i with maximum height will be accessed by any operation targeting a node with a higher key. However, N_i can later be deleted and inserted back with the minimum height; the operations that access it will then be extremely rare. The same reasoning holds when comparing an internal node with key K_i of a binary tree located at the root or close to the leaves.

As explained before, an accurate cache miss analysis cannot be satisfied with average access rates. Therefore, the information on the possible significant variations of rates should not be diluted into a single access rate of the node. To avoid that, we pass the information through virtual nodes: a node of the structure is divided into a set of virtual nodes, each of them holding a different flavor of the initial node (height of the node in the skip list or subtree size in the binary tree). The virtual nodes go through the whole analysis instead of the initial nodes, before we extract the average behavior of the system hence throughput.

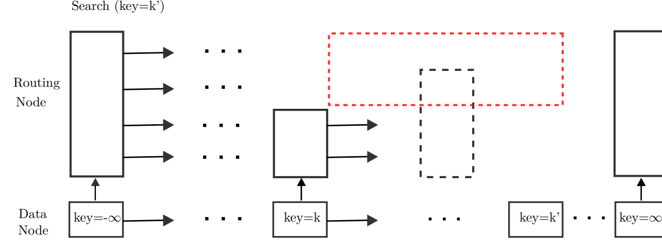


Figure 2 Skip List Events: Read Event Probability.

6.3 Skip List

There exist various lock-free skip list implementations and we study here the lock-free skip list [25].

Skip lists offer layers of linked lists. Each layer is a sparser version of the layer below where the bottom layer is a linked list that includes all the elements that are present in the search data structure. An element that is present in the layer at height h appears in layer at height $h+1$ with a fixed appearance probability (1/2 for our case) up to some maximum layer h_{max} that is a parameter of the skip list.

Skip list implementations are often realized by distinguishing two type of nodes: (i) valued nodes reside at the bottom layer and they hold the key-value pair in addition to the two pointers, one to the next node at the bottom layer and one to the corresponding routing node (could be *null*); (ii) routing nodes are used to route the threads towards the search key. Being coupled with a valued node, a routing node does not replicate the key-value pair. Instead, only a set of pointers, corresponding to the valued node containing the next key in different layers, are packed together in a single routing node (that fits in a cacheline with high probability). Every *Read* event in a routing node is preceded by a *Read* in the corresponding valued node.

We denote by $N_{k,h}^{rou}$ the routing node containing key k, whose set of pointers is of height h, where $h \in [1..h_{max}]$. A valued node containing the key k is denoted by $N_{k,h}^{dat}$ when connected to $N_{k,h}^{rou}$ ($h=0$ if there is no routing node). Furthermore, there are four sentinel nodes $N_{0,h_{max}}^{dat}$, $N_{0,h_{max}}^{rou}$, $N_{\mathcal{R}+1,h_{max}}^{dat}$, $N_{\mathcal{R}+1,h_{max}}^{rou}$. The presence probabilities result from the coin flips (bounded by h_{max}): for $z \in \{dat, rou\}$, $p_{k,h}^{z} = 2^{-(h+1)} q_k$ if $h < h_{max}$, $p_{k,h}^{z} = q_k - \sum_{\ell=0}^{h_{max}-1} p_{k,\ell}^{z}$ otherwise.

By decomposing into three cases, we compute the probability that an operation $op_{k'}^{o}$ of type $o \in \{ins, del, src\}$, targeted to k', causes a *Read* triggering event at $N_{k,h}^{z}$ when $N_{k,h}^{z} \in D$. Let assume first that $k' > k$. The operation triggers a *Read* event at node $N_{k,h}^{z}$ if for all (x,y) such that $y > h$ and $k < x \leq k'$, $N_{x,y}^{z}$ is not present in the skip list (*i.e.* in Figure 2, no node in the skip list overlaps with the red frame). Let assume now $k' < k$. The occurrence of a *Read* event requires that: for all (x,y) such that $y \geq h$ and $k' \leq x < k$, $N_{x,y}^{z}$, is not present in the structure. Lastly, a *Read* event is certainly triggered if $k' = k$. The final formula is given by:

$$\mathbb{P}\left[op_{k'}^{o} \rightsquigarrow read(N_{k,h}^{z})\right] = \begin{cases} \prod_{x=k+1}^{k'} \left(1 - \left(\sum_{y=h+1}^{h_{max}} p_{x,y}^{z}\right)\right), & \text{if } k \leq k' \\ \prod_{x=k'}^{k-1} \left(1 - \left(\sum_{y=h}^{h_{max}} p_{x,y}^{z}\right)\right), & \text{if } k > k' \end{cases}$$

To be complete, we describe in [4] how to compute the probability for *CAS* events, following a similar approach.

6.4 Binary Tree

We show here how to estimate the throughput of external binary trees. They are composed of two types of nodes: internal nodes route the search towards the leaves (routing nodes) and store just a key, while leaves, referred to as external nodes contain the key-value pair (valued node). We use the external binary tree of Natarajan [23] to instantiate our model. The search traversal starts and continues with a set of internal nodes and ends with an external node. We denote by N_k^{int} (resp. N_k^{ext}) the internal (resp. external) node containing key k, where $k \in [1..\mathcal{R}]$. The tree contains two sentinel internal nodes that reside at the top of the tree (hence are accessed by all operation): N_{-1}^{int} and N_0^{int}.

Our first aim is to find the paths followed by any operation through the binary tree, in order to obtain the access triggering rates, thanks to Equation 1. Binary trees are more complex than the previous structures since the order of the operations impact the positioning of the nodes. The random permutation model proposes a framework for randomized constructions in which we can develop our model. Each key is associated with a priority, which determines its insertion order: the key with the highest priority is inserted first. The performance characteristics of the randomized binary trees are studied in [24]. In the same vein, we compute the access probability of the internal node with key k in an operation that targets key k'.

▶ **Lemma 1.** *Given an external binary tree, the probability of accessing N_k^{int} in an operation that targets key $K_{k'}$ is given by: (i) $1/f(k,k')$ if $k' \geq k$; (ii) $1/(f(k',k)-1)$ if $k' < k$, where $f(x,y)$ provides the number internal nodes whose keys are in the interval $[x,y]$.*

Proof. N_k^{int} would be accessed if it is on the search path to the external node with key k'. Given $k' \geq k$, this happens iff N_k^{int} has the highest priority among the internal nodes in the interval $[k,k']$. This interval contains $f(k,k')$ internal nodes, thus, the probability of N_k^{int} to possess the highest priority is $1/f(k,k')$. Similarly, if $k' < k$, then N_k^{int} is accessed iff it has the highest priority in the interval $(k',k]$. Hence, the lemma. ◀

Even if in the binary tree, nodes are inserted and deleted an infinite number of times, Lemma 1 can still be of use. The number of internal nodes in the interval $[k,k']$ (or $(k',k]$ if $k' < k$) is indeed a random variable which is the sum of independent Bernoulli random variables that models the presence of the nodes. As a sum of many independent Bernoulli variables, the outcome is expected to have low variations because of its asymptotic normality. Therefore, we replace this random variable with its expected value and stick to this approximation in the rest of this section. The number of internal nodes in any interval come out from the presence probabilities: $p_k^z = q_k$, where $z \in \{int, ext\}$.

In an operation is targeted to key k', a single external node is accessed (if any): $N_{k'}^{ext}$, if present, else the external node with the biggest key smaller than k', if it exists, else the external node with the smallest key. Then, we have:

$$\mathbb{P}\left[op_{k'}^o \rightsquigarrow read\left(N_k^{int}\right)\right] = \begin{cases} 1/(1+\sum_{i=k'+1}^{k-1} p_i^{int}), & \text{if } k > k' \\ 1/(1+\sum_{i=k+1}^{k'} p_i^{int}), & \text{if } k \leq k' \end{cases},$$

$$\mathbb{P}\left[op_{k'}^o \rightsquigarrow read\left(N_k^{ext}\right)\right] = \begin{cases} 1, & \text{if } k = k' \\ \prod_{i=k+1}^{k'}(1-p_i^{ext}), & \text{if } k < k' \\ \prod_{i=1}^{k-1}(1-p_i^{ext}), & \text{if } k > k' \end{cases}$$

These probabilities finally lead to the computation of the *Read* (resp. *CAS*) rates $\lambda_{z,k}^{read}$ (resp. $\lambda_{z,k}^{cas}$) of N_k^z, where $z \in \{int, ext\}$, that will be used in the last following step.

We focus now on the *Read* rate of the internal nodes. We have found the average behavior of each node in the previous step; however, the node can follow different behaviors during the execution since the *Read* rate of N_k^{int} depends on the size of the subtree whose root is N_k^{int}, which is expected to vary with the update operations on the tree. We dig more into this and reflect these variations by decomposing N_k^{int} into H_k virtual nodes, $N_{k,h}^{int}$, where $h \in [1..H_k]$. We define the *Read* rate $\lambda_{int,k,h}^{read}$ of these virtual nodes as a weighted sum of the initial node rate thanks the two equations $p_k^{int} = \sum_{h=1}^{H_k} p_{k,h}^{int}$ and $p_k^{int} \lambda_{int,k}^{read} = \sum_{h=1}^{H_k} p_{k,h}^{int} \lambda_{int,k,h}^{read}$.

We connect the virtual nodes to the initial nodes in two ways. On the one hand, one can remark that the *Read* rate is proportional to the subtree size: $\lambda_{int,k,h}^{read} \propto h \lambda_{int,k}^{read}$. On the other hand, based on the probability mass function of the random variable Sub_k representing the size of the subtree rooted at N_k^{int}, we can evaluate the weight of the virtual nodes: $p_{k,h}^{int} = p_k^{int} \mathbb{P}[Sub_k = h]$.

More details are to be found in [4], on how to obtain the mass function of the random variable Sub_k to compute *Read* rates for the virtual nodes and how to deal with the *CAS* events.

7 Experimental Evaluation

We validate our model through a set of well-known lock-free search data structure designs, mentioned in the previous section. We stress the model with various access patterns and number of threads to cover a considerable amount of scenarios where the data structures could be exploited. For the key selection process, we vary the key ranges and the distribution: from uniform (*i.e.* the probability of targeting any key is constant for each operation) to zipf (with $\alpha = 1.1$ and the probability to target a key decreases with the value of the key). Regarding the operation types, we start with various balanced update ratios, *i.e.* such that the ratio of `Insert` (among all operations) equals the ratio of `Delete`. Then, we also consider asymmetric cases where the ratio of `Insert` and `Delete` operations are not equal, which changes the expected size of the structure.

7.1 Setting

We have conducted experiments on an Intel ccNUMA workstation system. The system is composed of two sockets, each containing eight physical cores. The system is equipped with Intel Xeon E5-2687W v2 CPUs. Threads are pinned to separate cores. One can observe the performance change when number of threads exceeds 8, which activates the second socket.

In all the figures, y-axis provides the throughput, while the number of threads is represented on x-axis. The dots provide the results of the experiments and the lines provide the estimates of our framework. The key range of the data structure is given at the top of the figures and the percentage of update operations are color coded.

We instantiate all the algorithm and architecture related latencies, following the methodologies described in [6] In line with these studies, we observed that the latencies of t^{cas} and t^{rec} are based on thread placement. We distinguish two different costs for t^{cas} according to the number of active sockets. Similarly, given a thread accessing to a node N_i, the recovery latency is low (resp. high), denoted by t_{low}^{rec} (resp. t_{high}^{rec}), if the modification has been performed by a thread that is pinned to the same (resp. another) socket. Before the execution, we measure both t_{low}^{rec} and t_{high}^{rec}, and instantiate t^{rec} with the average recovery latency, computed in the following way for a two-socket chip. For $s \in \{1, 2\}$, we denote by P_s the number of threads that are pinned to socket numbered s. By taking into account

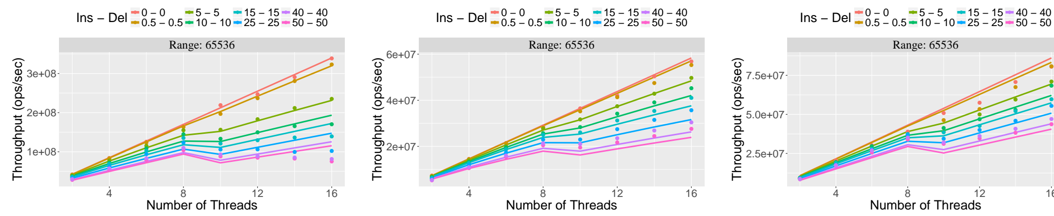


Figure 3 Hash Table with load factor 2.

Figure 4 Skip List.

Figure 5 Binary Tree.

all combinations, we have $t^{rec} = (P_1(P_1 t^{rec}_{low} + P_2 t^{rec}_{high}) + P_2(P_2 t^{rec}_{low} + P_1 t^{rec}_{high}))/P^2$. Since $P = P_1 + P_2$, we obtain $t^{rec} = t^{rec}_{low} + 2(P_1/P)(1 - P_1/P)(t^{rec}_{high} - t^{rec}_{low})$.

For the data structure implementations, we have used ASCYLIB library [11] that is coupled with an epoch based memory management mechanism which has negligible latency.

7.2 Search Data Structures

In Figure 3, 4 and 5, we provide the results for the hash table, skip list and binary tree where the key selection is done with the uniform distribution. One can see that the performance drops as the update rate increases, due to the impact of *CAS* related factors. This impact is magnified with the activation of the second socket (more than 8 threads) since the event becomes more costly. When there is no update operation, the performance scales linearly with the number of threads. Since the threads access disjoint parts of structure, we observe a similar behaviour for the cases with updates. See [4] for a more comprehensive set of experiments and applications.

8 Conclusion

In this paper, we have modeled and analyzed the performance of search data structures under a stationary and memoryless access pattern. We have distinguished two types of events that occur in the search data structure nodes and have modeled the arrival of events with Poisson processes. The properties of the Poisson process allowed us to consider the thread-wise and system-wise interleaving of events which are crucial for the estimation of the throughput. For the validation, we have used several fundemental lock-free search data structures.

As a future work, it would be of interest to study to which extent the application workload can be distorted while giving satisfactory results. Putting aside the non-memoryless access patterns, the non-stationary workloads such as bursty access patterns, could be covered by splitting the time interval into alternating phases and assuming a stationary behaviour for each phase. Furthermore, we foresee that the framework can capture the performance of lock-based search data structures and also can be exploited to predict the energy efficiency of the concurrent search data structures.

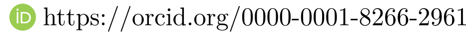
References

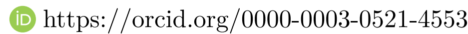
1 Dan Alistarh, Keren Censor-Hillel, and Nir Shavit. Are lock-free concurrent algorithms practically wait-free? In David B. Shmoys, editor, *STOC*, pages 714–723. ACM, June 2014.

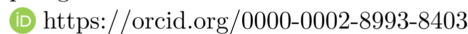
2 Aras Atalar, Paul Renaud-Goud, and Philippas Tsigas. Analyzing the Performance of Lock-Free Data Structures: A Conflict-Based Model. In *DISC*, pages 341–355. Springer, 2015.

3. Aras Atalar, Paul Renaud-Goud, and Philippas Tsigas. How Lock-free Data Structures Perform in Dynamic Environments: Models and Analyses. In *OPODIS*, pages 23:1–23:17, 2016.
4. Aras Atalar, Paul Renaud-Goud, and Philippas Tsigas. Lock-Free Search Data Structures: Throughput Modelling with Poisson Processes. *CoRR*, abs/1805.04794, 2018. URL: `http://arxiv.org/abs/1805.04794`.
5. Hagit Attiya and Arie Fouren. Algorithms adapting to point contention. *JACM*, 50(4):444–468, 2003.
6. Vlastimil Babka and Petr Tuma. Investigating Cache Parameters of x86 Family Processors. In *SPEC Benchmark Workshop*, pages 77–96. Springer, 2009.
7. A.D. Barbour and T.C. Brown. Stein's method and point process approximation. *Stochastic Process. Appl.*, 43(1):9–31, 1992.
8. Naama Ben-David and Guy E. Blelloch. Analyzing Contention and Backoff in Asynchronous Shared Memory. In *PoDC*, pages 53–62, 2017.
9. Timothy C. Brown, Graham V. Weinberg, and Aihua Xia. Removing logarithms from Poisson process error bounds. *Stochastic Process. Appl.*, 87(1):149–165, 2000.
10. Hao Che, Ye Tung, and Zhijun Wang. Hierarchical Web caching systems: modeling, design and experimental results. *IEEE Communications Society Press*, 20(7):1305–1314, 2002.
11. Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *ASPLOS*, pages 631–644. ACM, 2015.
12. Luc Devroye. A note on the height of binary search trees. *JACM*, 33(3):489–498, 1986.
13. Cynthia Dwork, Maurice Herlihy, and Orli Waarts. Contention in shared memory algorithms. *JACM*, 44(6):779–805, 1997.
14. James D. Fix. The set-associative cache performance of search trees. In *SODA*, pages 565–572, 2003.
15. Philippe Flajolet, Danièle Gardy, and Loÿs Thimonier. Birthday Paradox, Coupon Collectors, Caching Algorithms and Self-Organizing Search. *Discrete Applied Mathematics*, 39(3):207–229, 1992.
16. Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *PoDC*, pages 50–59. ACM, 2004.
17. Christine Fricker, Philippe Robert, and James Roberts. A versatile and accurate approximation for LRU cache performance. *CoRR*, abs/1202.3974, 2012.
18. Vincent Gramoli. More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *PPoPP*, pages 1–10. ACM, 2015.
19. Timothy L. Harris. A Pragmatic Implementation of Non-blocking Linked-Lists. In *DISC*, volume 2180 of *Lecture Notes in Computer Science*, pages 300–314. Springer, 2001.
20. Maurice Herlihy. A Methodology for Implementing Highly Concurrent Objects. *TOPLAS*, 15(5):745–770, 1993.
21. Peter Kirschenhofer and Helmut Prodinger. The Path Length of Random Skip Lists. *Acta Informatica*, 31(8):775–792, 1994.
22. John D. C. Little. A proof for the queuing formula: L= λ W. *Operations research*, 9(3):383–387, 1961.
23. Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *PPoPP*, pages 317–328. ACM, 2014.
24. Raimund Seidel and Cecilia R. Aragon. Randomized Search Trees. *Algorithmica*, 16(4/5):464–497, 1996.
25. Håkan Sundell and Philippas Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *J. Parallel Distrib. Comput.*, 65(5):609–627, 2005.

Concurrent Robin Hood Hashing

Robert Kelly
Maynooth University Department of Computer Science, Maynooth, Ireland
rob.kelly@cs.nuim.ie

https://orcid.org/0000-0001-8266-2961

Barak A. Pearlmutter
Maynooth University Department of Computer Science and Hamilton Institute,
Maynooth, Ireland
barak@cs.nuim.ie

https://orcid.org/0000-0003-0521-4553

Phil Maguire
Maynooth University Department of Computer Science, Maynooth, Ireland
pmaguire@cs.nuim.ie

https://orcid.org/0000-0002-8993-8403

Abstract

In this paper we examine the issues involved in adding concurrency to the Robin Hood hash table algorithm. We present a non-blocking obstruction-free *K-CAS* Robin Hood algorithm which requires only a single word compare-and-swap primitive, thus making it highly portable. The implementation maintains the attractive properties of the original Robin Hood structure, such as a low expected probe length, capability to operate effectively under a high load factor and good cache locality, all of which are essential for high performance on modern computer architectures. We compare our data-structures to various other lock-free and concurrent algorithms, as well as a simple hardware transactional variant, and show that our implementation performs better across a number of contexts.

2012 ACM Subject Classification Computing methodologies → Concurrent algorithms

Keywords and phrases concurrency, Robin Hood Hashing, data-structures, hash tables, non-blocking

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.10

Related Version `https://arxiv.org/abs/1809.04339`

Supplement Material `https://github.com/DaKellyFella/concurrent-robin-hood-hashing`

Funding Funded by the Government of Ireland Postgraduate Scholarship.

Acknowledgements I want to thank William Leiserson for his invaluable help with reviewing and feedback. I also want to thank Nir Shavit for extensive access to hardware for running the benchmarks.

1 Introduction

Concurrent data-structures allow multiple threads to operate on them without risk of data corruption, as well as providing guarantees of correctness for concurrent operations. Lock-free data-structures are a class of concurrent data-structures that have specific properties relating to system or thread progress guarantees. The programming of portable and practical lock-free

© Robert Kelly, Barak A. Pearlmutter, and Phil Maguire;
licensed under Creative Commons License CC-BY

data-structures is becoming ever more practical, with the addition of mainstream language support for atomic variables, and a well defined thread memory model (see [5], [28]).

Concurrent algorithms can be separated into two major classes, namely blocking and non-blocking, with both featuring further partitions based on the specific progress guarantees within those classes [20]. Blocking algorithms have well documented issues when it comes to their use. They are susceptible to deadlock, priority inversion, convoying, and a lack of composability with respect to multiple operations on data-structures. A lock-free, or non-blocking, algorithm has none of these problems. Such algorithms suffer, however, from their own set of challenges relating to memory management ([1], [2], [26], [14], [9]), correctness ([22], [24], [3]) and potentially lackluster performance as the system is flooded with contention under heavy write load. As hardware manufacturers resort to expanding processor core counts for enhanced performance [34], non-blocking data-structures are coming to the fore, providing more robust progress guarantees and tolerance to the suspension of threads. For end consumers to realise the full performance of their system, algorithms must efficiently exploit as many cores as possible.

Hash tables are one of the major building blocks in software applications, providing efficient implementations for the abstract data types of maps and sets. These data-structures are highly versatile, making them an active area of research in concurrency (e.g. [25], [20], [21], [8], [27], [30], [32]). Hash tables are associative data-structures that contain a pool of keys and associated values [10], lending themselves to efficient implementations. In general, they feature the methods `Add`, `Remove`, and `Contains`, each of which is bound by $\mathcal{O}(1)$ computational complexity while requiring $\mathcal{O}(n)$ space [10]. Hash table algorithms achieve this performance by calculating an index, called a *hash*, from each key, and use this to efficiently find the relevant entry in the pool of keys, normally an array. Unlike comparative structures, which store keys in a sorted order and binary search through the space, hash tables rely on the hash function to distribute the keys across the space. Ideally, the hash function generates a unique index for each key. In reality, however, the keys often have the same hash, creating what is known as a *collision*. A primary focus of research in hash tables is how to efficiently deal with these collisions.

Hash tables can be divided into two major design variants, namely open-addressing and closed-addressing (i.e. separate-chaining). Open addressing stores the key and value pairings in different buckets in the table, either through a pointer or directly in the internal array itself, with a single item allowed per bucket. When a hash collision takes place (when another entry has taken the desired bucket), a new bucket is selected via some collision resolution algorithm. Separate chaining, on the other hand, stores a pointer to a list of values at that bucket, containing all the key and value pairings that collided on that bucket.

Robin Hood Hashing [7] is an open-addressing hash table method in which entries in the table are moved around so that the variance of the distances to their original bucket is minimised. Insertion is a multi-stage process, potentially moving multiple items throughout the table. The general goal is to find an empty bucket, or another entry that is less 'deserving' of its bucket than the current item being inserted or moved. If an empty bucket is found, it is taken. If another less deserving entry has been identified, then it's swapped with the current entry and 'kicked' further down the line, with this process repeating until an empty bucket is found. The serial version of Robin Hood is well suited to modern computer architecture. As CPU utilisation effectively becomes a function of the memory bottleneck [11], algorithms that use the CPU cache more efficiently can enhance performance for memory bounded tasks. For instance, Robin Hood Hashing has a very low expected probe count, allowing reads to be culled early even though the algorithm uses linear probing. Low probe counts mean fewer

cache misses, performing very well on modern architectures. As it stands, these attractive properties have been maintained in our concurrent versions. Our concurrent solution manages to achieve non-blocking progress with physical deletion and all of the aforementioned benefits of the serial algorithm. We use an algorithmically optimised *K-CAS* [4] implementation along with a timestamp mechanism to handle bulk relocations while maintaining correctness.

In Section 2 we review the current landscape of concurrent and lock-free hash tables, and the original Robin Hood algorithm. Section 3 outlines the structure of our algorithms and the various challenges encountered in adding concurrency, while Section 4 discusses the performance of these algorithms relative to competitors.

2 Background

2.1 Prior Work

A number of concurrent open-addressing hash table algorithms have been proposed. Purcell and Harris [30] presented a lock-free open-addressing hash table where per-bucket upper bounds are stored in conjunction with the keys, thus allowing searches to be culled early. Nielson and Karlsson [27] built upon the work of Purcell and Harris with a Lock-Free Linear Probing hash table, simplifying the earlier algorithm, reducing the number of bucket states required, and removing the word normally required. Herlihy, Shavit, and Tzafrir [21] presented Hopscotch Hashing, a concurrent hash table algorithm with outstanding performance. This algorithm allows searches, insertions, and removals to skip over irrelevant items, and is also cache aware in its reordering of entries present in the table. While Hopscotch's insertions and deletions are blocking, the mutating operations are *sharded* over multiple locks.

Like open-addressing, separate-chaining can be implemented in many different forms. Michael [25] presented a lock-free hash table, with each bucket containing a slightly modified Harris linked list [16]. Shalev and Shavit [32] presented a particularly succinct implementation of using a linked list as a hash table, indexing into it from an array of node pointers. In this case, the list can grow forever, so long as pointers to new entry points are added to prevent the probe length from growing out of control. The table can be automatically resized, as only the entry point array of pointers need be extended. Laborder, Feldman, and Dechev [13] presented their Wait-Free Hash Table, whereby a key collision at a bucket leads the bucket to be expanded into another sub-table until a point is reached such that all collisions are resolved.

2.2 Original Robin Hood

Robin Hood Hashing was first proposed by Celis [7] in 1986. Though remaining relatively obscure, it has recently gained recognition via the new programming language Rust [35], which has adopted Robin Hood as its standard hash table algorithm. Robin Hood is an open-addressing hash table algorithm that employs linear probing for finding items and for finding spaces for new entries. Robin Hood does exactly what it says on the tin: it steals from the rich and gives to the poor. Here, "rich" refers to items that got lucky during the hashing process, by finding a free bucket close to their original bucket. In other words, these "rich" items have a low **D**istance **F**rom their expected **B**ucket (this distance is referred to as *DFB* for short) and a low expected probe count before being found. In contrast, being "poor" means hashing to a bucket that has been heavily saturated beforehand, and thus has a high number of items to step over before finding a free bucket. "Poor" items thus have a higher *DFB*, and a high expected probe count before being found.

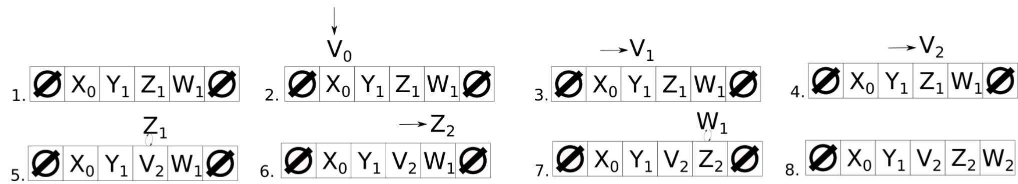


Figure 1 An example Robin Hood insertion where `V` is inserted into `X`'s bucket. Each step of the insertion is numbered.

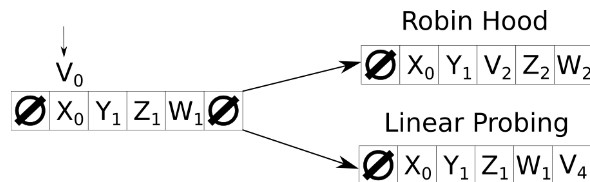


Figure 2 Comparison of Robin Hood to Linear Probing for insertion of `V`.

Robin Hood solves this inequality during insertion by moving existing entries around the table. In other words, when the item being relocated has a larger *DFB* than the item currently being examined, these items are swapped, and the search continues for an empty bucket. Once an empty bucket is found, the relocated item is inserted, and the process terminates. Figure 1 shows an example Robin Hood insertion. In all examples the subscripts represent the *DFB* of each entry. Step 1 shows the table initially containing `X`, `Y`, `Z`, and `W`, each with a *DFB* of 0, 1, 1, and 1 respectively. Step 2 shows that `V` is to be inserted where `X` currently resides. Step 3 shows how `V` doesn't kick out `X`, as they're equal in *DFB*; the same is true for `Y` in step 4. Step 5 shows the swap between `V` and `Z`, as `V` is now further away than `Z` (*DFB* of 2, compared to 1). `Z` linear probes further down the table in step 6, and in step 7 swaps with `W`. Finally, in step 8 `W` lands in the empty bucket at the end, and the insertion process finishes. In summary, `V` is swapped with `Z`, `Z` is swapped with `W`, and, finally, `W` is placed in the empty bucket at the end. Figure 2 shows a comparison of the Robin Hood insertion and the same insertion using the Linear Probing collision resolution. As can be seen, the entry `V` ends up far further away using Linear Probing than using Robin Hood.

The outcome of this shuffling is a reduced average variance in *DFB* (i.e. reduced probe lengths). Not only does reduced variance make for more predictable and uniform performance, it also allows searches to be culled early, without having to find an empty bucket, which is typically the requirement for calling off a linear probing search. As soon as the item being examined during the search has a lower *DFB* than the current probing *DFB* the search can be called off, with the knowledge that it will be unsuccessful: the item being searched for cannot possibly be present in the table, as it would already have kicked out any items with a lower *DFB* than itself. Figure 3 shows an example Robin Hood search operation. The key `U` is being queried, probing the table as far as the bucket containing `Z` before terminating. The linear probing is shown in steps 1 to 3, while termination happens at step 4. Termination is possible as `U` is 3 buckets away from its original bucket, whereas `Z` is only 2, meaning `U` can't possibly be in the table. The reasoning is that if `U` was being inserted it would have displaced `Z` for having a higher *DFB* than itself; therefore, it cannot possibly be in the table. A similar search taking place using linear probing would have to keep searching until an empty bucket is found, resulting in far more probes at higher load factors, and higher amounts of cache misses too. We refer to this search mechanism as the Robin Hood invariant. Violating this invariant leads to a corrupted table, potentially losing items in the table. These factors

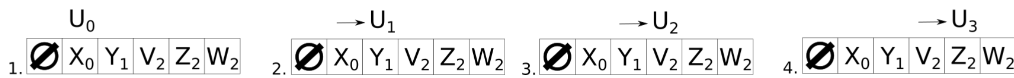


Figure 3 Example Robin Hood search, querying the table for the entry U.

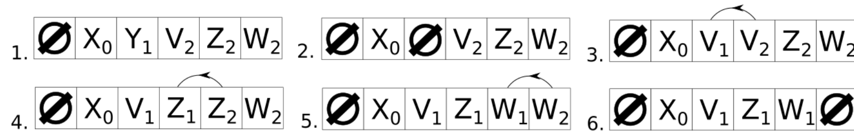


Figure 4 Example Robin Hood deletion. Entry in question is Y.

enable Robin Hood to support a higher load factor, given that the expected search is only 2.6 probe counts on average for successful searches, and $\mathcal{O}(ln(n))$ for unsuccessful searches [7].

Deletion in Robin Hood is more complicated than in linear probing. In linear probing one can simply tombstone a bucket, as it does not matter what entry was there before. However, in Robin Hood the *DFB* of the entry matters, as subsequent searches employ this metric to determine if the entry being queried is contained in the table. The solution to this is to either logically mark the entry as deleted, or shift back every entry in front of it until some criterion is met. Logical deletion is unacceptable, as it causes the table to fill up and lose its efficiency, resulting in unnecessary resizes. Backward shifting effectively undoes the insertion of the entry we wish to delete from the table. An example of this is given in Figure 4. Since the removing thread cannot tell which entries the item to be deleted originally displaced to get there (assuming they aren't at their ideal bucket), we must shift back all items. The shifting is continued until we find an empty bucket or an item in its ideal bucket. Step 1 shows the initial table, step 2 shows the "nulling" of `Y`. Steps 3, 4, and 5 show a backward shifting of entries within the table. Step 6 shows the termination of the shifting, as an empty bucket is ahead of `W`.

2.3 K-CAS

K-CAS or, multi-word-compare-and-swap, is an extended version of *CAS*, supporting multiple compare-and-swap operations on many distinct memory locations, all of which either succeed or else fail together. Our algorithm employs a modified version of the *K-CAS* originally proposed by Harris, Kaiser and Pratt [17]. The *K-CAS* algorithm does come at a small cost, reserving an additional 0-2 bits for each word being manipulated by the algorithm. These reserved bits are needed to store run-time type information that allows descriptors to be distinguished from normal values. The standard *K-CAS* interface provides two functions for basic reading and writing, `K_CAS_READ(loc: T*) -> T` and `K_CAS_WRITE(loc: T*, T val) -> void`, and a mechanism for adding addresses and their values to a descriptor. The rationale for needing dedicated read and write functions is that the values being operated on have specific bits reserved to indicate an ongoing *K-CAS* operation. Both the read and write functions help any pending *K-CAS* operation installed at that particular memory location.

Traditional *K-CAS* implementations need a memory reclaimer system, as each descriptor must be fresh to avoid the ABA problem [29] . However, the specific *K-CAS* implementation we use, developed by Arbel-Raviv and Brown [4], employs descriptor reuse, thereby eliminating the need for a freshly allocated descriptor for each operation. Given that this implementation does not require a memory allocation per *K-CAS* operation, or a reclaimer, its performance is substantially improved. This enhancement makes *K-CAS* a feasible concurrency primitive and, as we show, allows it to outperform the best lock-based hash table algorithms.

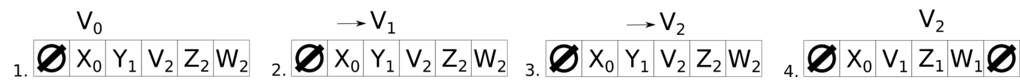


Figure 5 The problem of concurrent searches and removals. The searcher is trying to find `V` while entry `Y` is being removed.

3 Algorithm

3.1 Challenges For Concurrent Robin Hood

The primary challenge in making Robin Hood concurrent is, unsurprisingly, the modifying operations on the table. Both `Add` and `Remove` can modify large parts of the table, with `Add` potentially performing a global table reorganisation, and `Remove` potentially shifting back many entries. These problems defeat naive solutions such as *sharded* locks, as `Add` could potentially grab all of them, leading to no concurrency. Another issue is that of deadlock; insertions at two different points in the table might grab the locks in a cyclic manner, leading to deadlock. For example, the table could have 8 locks with 8 ongoing insertions in 8 different locations. Initially, each insertion will grab one lock corresponding to the original location. If all insertions relocate an entry to another lock section, deadlock occurs. To achieve a truly concurrent implementation of Robin Hood we need an efficient mechanism to update large disparate parts of the underlying table. For this there are two options. The first is *K-CAS*, which became feasible from a performance standpoint thanks to the work of Arbel-Raviv and Brown [4]. The second choice is hardware transactional memory, which we use to provide efficient speculative lock-elision [31].

3.2 Overview

K-CAS is a natural choice for relocation-based hash table algorithms. It prohibits a number of issues that typical concurrent algorithms run into, for example, examining whether invariants hold during some intermediate operation or state. *K-CAS* behaves like an expressively weaker transactional memory [19], but unlike hardware transactional memory [37], it has well defined progress guarantees. Another reason for using *K-CAS* is that keys can be stored directly in the table, thus improving cache locality. The entry relocations initiated by modifications are summarised into a *K-CAS* descriptor instead of relying on in-place modifications of the table featuring entry relocation information. Threads cannot see a *K-CAS* operation partially completed. Nevertheless, special considerations need to be made for operations when reading the table, as they can experience inconsistent views if applied naively.

Since entries can be moved around during concurrent reads, they could inadvertently miss a key due to some ongoing relocation operation. For example, when `Contains` uses the Robin Hood invariant to terminate a search early, there is a race with a concurrent `Remove` which could have shifted that particular entry in question back through the table, behind the reader, leading to an incorrect result. This happens when `Remove` is called on an unrelated entry located in the vicinity of the entry being queried. An example of this phenomenon is illustrated in Figure 5. Here the entry `V` is being queried while at the same time entry `Y` is being removed. When the reader gets to the bucket containing `V`, the remover executes its *K-CAS* operation, shifting a number of entries, including `V`, backwards. The searcher then checks the bucket after the shift and sees `Z`, terminates the search, and falsely declares `V` as not present in the table.

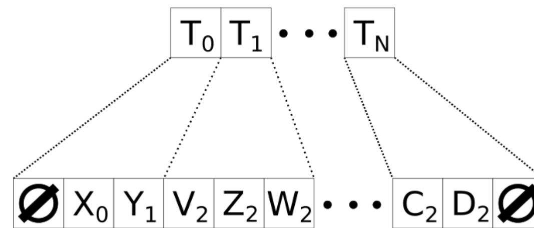


Figure 6 An illustration of how the timestamps are *sharded* across multiple entries.

Our solution to avoiding this race is to associate each part of the table with a timestamp, updated upon every relocation. Figure 6 shows the correspondence of timestamps to physical buckets in the table. A timestamp can be mapped onto several buckets in the table. The mapping of the timestamps is identical to how locks are *sharded* in blocking hash tables like Hopscotch Hashing [21]. When a reader is checking for the presence of a particular key, the reader remembers the timestamps encountered during its search. If the key is found and read atomically, then the search can finish. However, if they key isn't found the reader must check the values of those timestamps after the search has completed. If a discrepancy is found, the search is restarted, otherwise we know for certain the key isn't in the table. When `Remove` is executing, it increments the timestamp every time it shifts an entry, and similarly for the `Add` method.

3.3 Algorithm Methods

We now outline our algorithm with annotations of code presented in Figures 7, 8, and 9, which highlight and explain important parts and what they do. The algorithmic code has been simplified in two areas. The first simplification involves timestamps: the code provided doesn't check if a timestamp has already been added to the list, nor does it check the number of entries per timestamp. The second simplification involves `Remove` when shuffling elements back. In both cases the code is long but simple, so we have excluded it for the sake of clarity.

A - Contains

All line numbers refer to Figure 7. At the beginning of the `Contains` method, a timestamp list is created. Line 8 adds the timestamp for that bucket to the list of timestamps. Line 9 loads a candidate key from the table; if it's a `Nil` key, then the search is culled and the timestamps are checked. Line 11 handles a matching key, returning `true`. Lines 12 – 13 handle the case where the distance of the key is too far away from its original bucket to be present in the table. Lines 17 – 19 check if the timestamp for the particular key has changed since the beginning of the operation, requiring the entire `Contains` method to be retried, as a key could have been missed.

B - Add

All line numbers refer to Figure 8. `Add` is quite similar to the serial version. Lines 4 – 5 keep track of the entry currently being relocated via the active key, initially setting the variable to the key being inserted. `Add` also keeps track of the last timestamp bucket it has incremented. This is hidden behind a helper function called `add_timestamp_increment`. Timestamps are incremented to prevent a concurrently running `Add` or `Remove` from interfering with the correctness of the method. Lines 10 – 14 attempt to insert the key currently being relocated

```
1  fn Contains(key: K) -> bool {
2    start_bucket: u64 = hash(key) % size;
3  retry:
4    timestamps: List<u64> = [];
5    for(i = start_bucket, cur_dist = 0;; cur_dist++, i++) {
6      i %= size;
7      // Add the timestamp for that specific index.
8      timestamps.append(read_timestamp(i));
9      cur_key: K = K_CAS_load(&table[i]);
10     if(cur_key == Nil) { goto timestamp_check; }
11     if(cur_key == key) { return true; }
12     distance: u64 = calc_dist(cur_key, i); // Robin Hood Invariant
13     if (distance < cur_dist) { goto timestamp_check; }
14   }
15 timestamp_check:
16   // Compare every timestamp.
17   for(i = start_bucket, idx = 0; idx < timestamps.size(); i++, idx++) {
18     i %= size;
19     if(list[idx] != read_timestamp(i)) { goto retry; }
20   }
21   return false; // No key
22 }
```

Figure 7 Pseudo-code for `Contains`.

into a `Nil` bucket, retrying the whole `Add` operation on failure. Line 16 returns `false` upon a key match, as the entry is already in the set. Lines 17 – 24 check if an entry needs to be relocated, replacing it with the entry currently being relocated in the thread's *K-CAS* descriptor.

C - Remove

All line numbers refer to Figure 9. `Remove` is a combination of `Contains` and `Add`. First, `Remove` tries to find the key. If the key isn't found then, as per `Contains`, timestamps are checked in case a concurrent `Remove` or `Add` has relocated the key during its search. If a key is found, then the process of deletion begins. Lines 12 – 17 constitute the deletion process. The function `shuffle_items` linearly shuffles items back until a `Nil` key is found, or else an entry with a *DFB* of 0 is found. As mentioned earlier, the function is simple, though expansive, so we exclude it. Each linear shuffle is put into the *K-CAS* descriptor, with line 15 performing the *K-CAS* operation. The ultimate outcome of this operation is the physical deletion of the entry. Lines 19 – 20 check for the Robin Hood invariant, terminating the search and checking timestamps if the criterion is met. Like `Contains`, `Remove` will restart if there is a discrepancy in the timestamps. Lines 24 – 26 check the timestamps of each bucket read.

```
1  fn Add(key: K) -> bool {
2    start_bucket: u64 =  hash(key) % size;
3  retry:
4    active_key: K = key;
5    descriptor: K_CAS_Desc = create_descriptor();
6    for(i = start_bucket, active_dist = 0;; i++, active_dist++) {
7      i %= size;
8      cur_timestamp = read_timestamp(i);
9      cur_key: K = K_CAS_load(&table[i]);
10     if(cur_key == Nil) {
11       descriptor.add(&table[i], Nil, active_key);
12       res: bool = K_CAS(descriptor); // Attempt to K-CAS operations
13       if(!res) { goto retry; }
14       return true;
15     }
16     if(cur_key == key) { return false; }
17     distance: u64 = calc_dist(cur_key, i); // Robin Hood Invariant
18     if (distance < active_dist) {
19       descriptor.add(&table[i], cur_key, active_key); // Swap keys
20       // Increment the timestamp within the descriptor.
21       add_timestamp_increment(i, cur_timestamp);
22       // Swap active key and kick this key down the table
23       active_key = cur_key;
24       active_dist = distance;
25     }
26   }
27 }
```

Figure 8 Pseudo-code for `Add`.

3.4 Proof Of Correctness

The proof of correctness is relatively simple and informal. *K-CAS* [17] is itself *linearisable* [22], and *K-CAS* encodes all relocations and timestamp increments to the data structure in its descriptors. An invocation of *K-CAS* essentially turns each modification operation on the table into a transaction. Every `Add` or `Remove` call that results in relocations increments a timestamp via the *K-CAS* operation; if any reading thread was to examine the table's contents during a relocation, the reader could compare before and after timestamps so as to ensure that no entry was moved during the search. Furthermore, because each reader also helps the *K-CAS* operation, reading the timestamp will result in one of three outcomes. Either the reader will read a new timestamp value, the same timestamp, or else help the operation complete an ongoing *K-CAS* operation by reading the new timestamp if the operation succeeds or the old one if it fails. Readers need only ensure that the timestamps haven't changed since their initial reading. Since `Remove` can exit in two ways, it has two different *linearisation* points. The *linearisation* point of `Add`, and the first code path of `Remove` are at line 12 in Figure 8 and line 15 in Figure 9, where the *K-CAS* is successfully called. `Contains` and the second exit point of `Remove` *linearise* at the point of reading their last timestamp from their search on lines 8 and 9 respectively.

```
1  fn Remove(key: K) -> bool {
2    start_bucket: u64 = hash(key) % size;
3  retry:
4    timestamps: List<u64> = [];
5    descriptor: K_CAS_Desc = create_descriptor();
6    for(i = start_bucket, cur_dist = 0;; cur_dist++, i++) {
7      i %= size;
8      // Add the timestamp for that specific index.
9      timestamps.append(read_timestamp(i));
10     cur_key: K = K_CAS_load(&table[i]);
11     if(cur_key == Nil) { goto timestamp_check; }
12     if(cur_key == key) {
13       // Shuffle items down until a Nil key or dist(key) == 0
14       shuffle_items(i, cur_key);
15       res: bool = K_CAS(descriptor);
16       if(!res) { goto retry; }
17       return true;
18     }
19     distance: u64 = calc_dist(cur_key, i); // Robin Hood Invariant
20     if (distance < cur_dist) { goto timestamp_check; }
21   }
22 timestamp_check:
23   // Compare every timestamp.
24   for(i = start_bucket, idx = 0; idx < timestamps.size(); i++, idx++) {
25     i %= size;
26     if(list[idx] != read_timestamp(i)) { goto retry; }
27   }
28   return false;
29 }
```

Figure 9 Pseudo-code for `Remove`.

3.5 Progress

The progress of each operation is parameterised by the progress of the *K-CAS* operation. If the *K-CAS* operation is blocking, then all method calls on the hash table are blocking. However if we have a lock-free implementation of *K-CAS*, then the following classifications for each method occur. Calls to `Contains` are obstruction-free [18], as other concurrent operations can cause a relevant timestamp to change, thus forcing the method to restart. Threads calling `Contains` can starve but they are not blocked if another thread dies. `Add` has the same progress guarantees as `Contains` and for the same reasons, other modifying operations can modify relevant timestamps forcing the method to fail and restart. Timestamps are not guaranteed to correspond to the entries the method wants to examine, they are coarse and each timestamp corresponds to a number of entries. `Remove` must first find the entry it wishes to remove, effectively running the same code as `Contains` and thus having the same classification. Afterwards, once found, the deletion and subsequent shuffling of that entry is also obstruction-free since other irrelevant operations with the same timestamp can interfere with its progress. In summary, lock-free *K-CAS* calls to `Contains`, `Add`, and `Remove` are obstruction-free.

4 Performance, Results, and Discussion

In this section we detail the performance and implementation of our algorithms. All of our code is made freely available online [23]. This includes *K-CAS* Robin Hood, the hardware transactional [19] variant of Robin Hood Hashing with lock-elision [31], the implementation of alternative competing algorithms (either coded by us or obtained via online sources), and benchmarking code which allows readers to replicate our results.

4.1 Experimental Setup

For our experiments we opted to use a set of microbenchmarks which stressed the hash table under various capacities and workloads. Our benchmarks were run on a 4 CPU machine, with each CPU (Intel® Xeon® CPU E7-8890 v3) featuring 18 cores with 36 hardware threads and 512 GiB RAM. The machine was running Ubuntu 14.04 with kernel version 3.13.0-141. Each thread was pinned to a specific core during testing. When scaling the number of threads, care was taken to pin the thread to a new core, avoiding *HyperThreading™* until necessary. Once all non *HyperThreading™* cores on each CPU were exhausted, *HyperThreading™* was employed thereafter. The rational for this scheduling choice is that algorithms which employ hardware transactional memory are disproportionately penalised by *HyperThreading™* and some mainstream operating systems, such as OpenBSD, disable *HyperThreading™*. As a result we shed light on how our algorithms would scale on those systems. *NUMA* memory effects were controlled by specifying where each thread could allocate using the `numactl` command, allocating on the RAM banks closest to the running CPUs as they came into use.

The algorithms used in the experiments are as follows: *K-CAS* Robin Hood, Transactional Lock-Elision Robin Hood Hashing, Hopscotch Hashing [21], a Lock-Free Linear Probing hash table described by Nielsen and Karlsson [27], and Michael's lock-free hash table [25].

A number of workload configurations were used in graphing the results. Four load factors of 20%, 40%, 60%, and 80% were chosen, along with two update workload configurations, namely 10% and 20%, referred to herein as "light" and "heavy". Both of the workload configurations were tried at the specified load factors, and both were used for comparing the different Robin Hood Hashing algorithms against each other, and against competitors. We sized the tables at 2^{23} to ensure that they wouldn't fit into the cache, thereby exposing each algorithm's effective cache use. The key space was equal to the size of the table, and was filled to the specified load factors. The scalable JeMalloc [12] allocator was used in the experiments and no memory reclamation system was used in algorithms that traditionally require one.

The testing process was carried out as follows: Each thread calls a random method with a random argument from some predefined method and key distribution. All threads are synchronised before execution on the data structure, and for a specified amount of time rather than a specific number of iterations. Each thread counts the number of operations it performed on the structure during the benchmark. The total amount of operations per microsecond for all threads is then graphed. Each experiment was run five times for 10 seconds each, and the average of each result was computed and plotted. All of our algorithms were written in C++11, and compiled with *g++* 4.8.4 with `O3` level of optimisation. Cache misses were collected via PAPI [36].

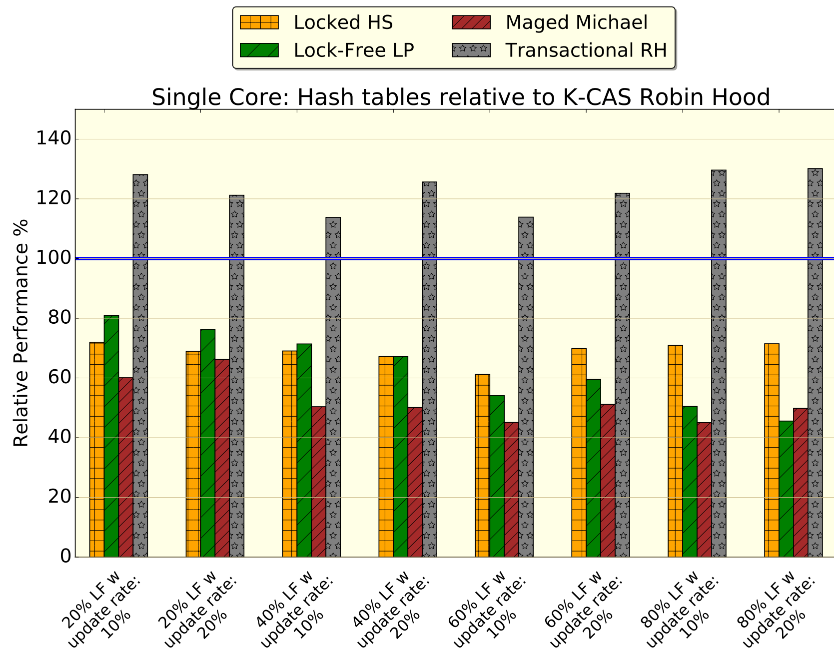


Figure 10 Single-core performance of the hash tables relative to *K-CAS* Robin Hood.

Table 1 Cache misses relative to *K-CAS* Robin Hood single-core.

Hash Tables	Configurations (Load factor w/ Updates)							
	20% w/ 10%	20% w/ 20%	40% w/ 10%	40% w/ 20%	60% w/ 10%	60% w/ 20%	80% w/ 10%	80% w/ 20%
Hopscotch Hashing	88%	82%	70%	71%	64%	68%	59%	65%
Lock-Free LP	185%	207%	227%	240%	294%	305%	430%	453%
Maged Michael	109%	105%	95%	95%	89%	93%	85%	91%
Transactional RH	86%	85%	82%	78%	89%	86%	93%	90%

4.2 Discussion and results

In order to gain an understanding of general operation overhead, we measured single-core performance. The performance is measured against *K-CAS* Robin Hood. In Figure 10 we see that Hopscotch Hashing, Maged Michael's Separate Chaining, and Lock-Free Linear Probing are significantly slower than the other algorithms. The reason for this is two-fold. First, the issue of cache efficiency arises: Lock-Free Linear Probing and separate chaining use dynamic memory allocation, meaning that a pointer dereference is needed for every bucket access. While Hopscotch Hashing does not use dynamically allocated memory, it does put more pressure on the cache by storing the original hash of a key inside the table. However it should be noted that it doesn't put as much pressure on the cache as *K-CAS* Robin Hood, as per Table 1. The second issue is the amount of work carried out for every operation: Hopscotch Hashing is the most complicated, executing more code and performing more operations. Transactional Robin Hood has higher performance than *K-CAS* Robin Hood as it does not require to consult the *K-CAS* descriptor and timestamps.

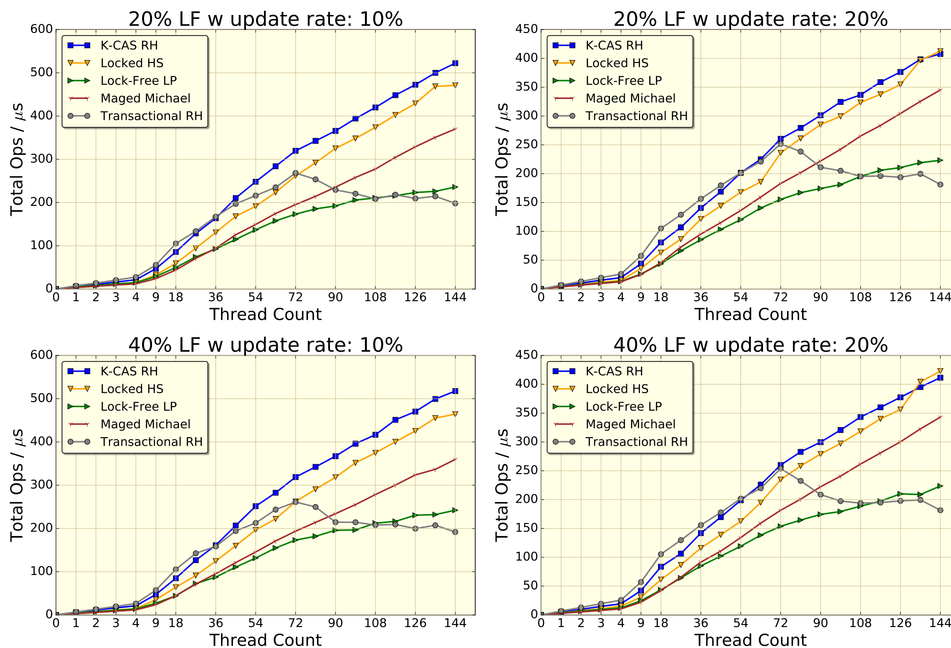


Figure 11 Cumulative operations per microsecond for the hash tables at 20% and 40% load factors at two update rates. (Higher is better)

The cache results can be seen in Table 1 as a percentage relative to *K-CAS* Robin Hood for a single-core. These cache statistics were measured over the course of the entire execution of the benchmark for each table and for each configuration. Hopscotch Hashing fares very well as it is able to skip over irrelevant entries. Lock-Free Linear Probing uses dynamic memory and thus puts enormous pressure on the cache. Another downside is that as the table fills up over time with tombstones, it forces operations to take roughly the same amount of time regardless of load factor. This phenomenon is called *contamination* [15]. Maged Michael [25] fares reasonably well even though it uses dynamic memory. Very few buckets have more than a single node, meaning few extra nodes are needed. Transactional Lock-Elision Robin Hood performs better as it does not need to consult an extra timestamp array or any extra *K-CAS* descriptor, which would require an extra level of indirection.

As is clear from the multi-core results in Figure 11, 12 the *K-CAS* Robin Hood Hashing algorithm either scales better than, or is competitive with, its opposition. Across all graphs there are two significant dips in performance. The first is at 18 threads, where threads are pinned to a different CPU socket. The use of another socket requires inter-socket communication and *NUMA* effects, reducing overall performance. The second is when increasing from 72 to 81 threads, which is the point where *HyperThreading™* kicks in. This kink is particularly pronounced for the transactional Robin Hood variant, which never recovers after that point. Both of these effects become most pronounced when a significant write load is placed on the tables.

Each of the workloads highlights the particular performance characteristics of each hash table. The update-light (10% writes) workload shows *K-CAS* Robin Hood dominating the competition across all thread counts and load factors. The update-heavy (20% writes) workload has two distinct outcomes predicated on the table load factor. The lower load factor shows *K-CAS* Robin Hood is slightly beaten out by the Transactional Robin Hood

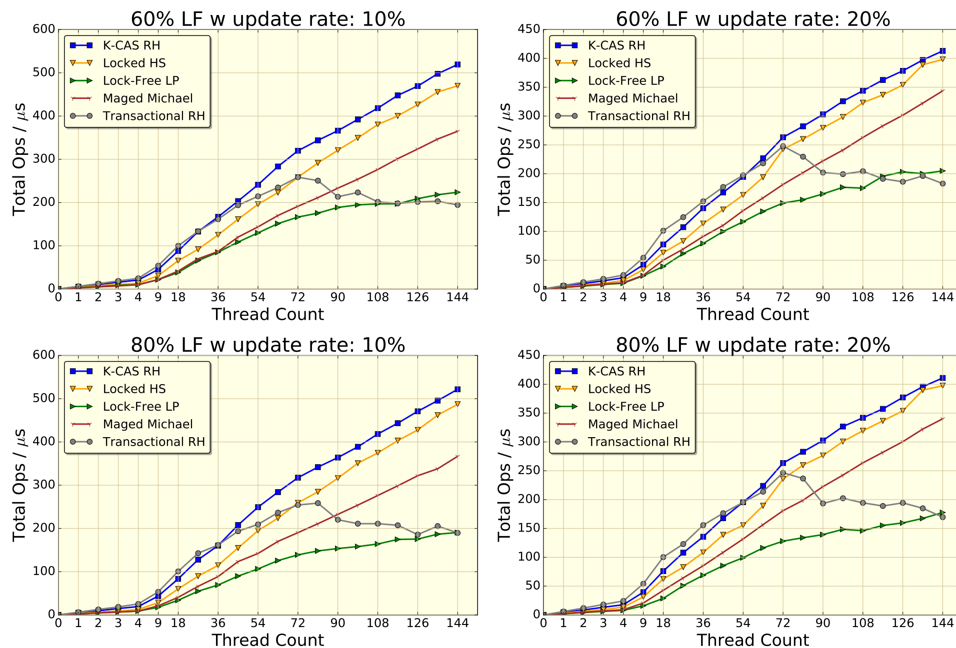


Figure 12 Cumulative operations per microsecond the hash tables at 60% and 80% load factors at two update rates. (Higher is better)

variant until the 72 thread mark, wherein it drops off entirely. *K-CAS* Robin Hood also edges out Hopscotch Hashing throughout most of the update-heavy workload until the very upper end of the thread count, where it is slightly outclassed. However, at higher load factors *K-CAS* Robin Hood demonstrates similar relative performance to the lower load factor benchmarks but maintains its lead over Hopscotch Hashing. All workloads show the gap between *K-CAS* Robin Hood and Hopscotch begins to narrow once *HyperThreading™* kicks in. As mentioned earlier, in every configuration the transactional variant of Robin Hood scales very well until 72 threads, after which *HyperThreading™* causes a huge kink in performance, and the algorithm never recovers. Maged Michael scales well in the tested workloads, though the gradient of the line isn't steep enough to challenge *K-CAS* Robin Hood or Hopscotch Hashing. Lock-Free Linear Probing does the worst of all, ending up at the same end point as Transactional Robin Hood.

4.3 Future Work

An issue we don't deal with is *resize*, specifically, when to resize the table and how to do it. Lock-free *resize* methods have been discussed in the literature ([8], [32]). To the best of our knowledge there has not yet been a formally published generic hash table resize method. Another item for future work is a combination of lock-free *K-CAS* and transactional [19] memory, such as the algorithm presented by Trevor Brown [6] for lock-free trees. A similar exploration for Robin Hood and other the hash tables benchmarked would be of interest. Work done by Siakavaras et el. [33] shows that a naive application of hardware transactional systems to data-structures typically perform poorly and require algorithm modifications to operate efficiently. Future work aimed at determining the optimal modifications for Robin Hood might elevate its performance for hardware transactional memory.

5 Conclusion

We have presented an obstruction-free Robin Hood Hashing algorithm which achieves superior performance relative to other concurrent algorithms with similar capabilities, and a hardware transaction variant which demonstrates best in class performance. These results establish that the Robin Hood algorithm is algorithmically suited to concurrency. Our experiments have shown that it scales strongly at all thread counts. Unlike linear probing, the algorithm can scale effectively to significantly higher load factors. It is very simple in nature, relying primarily on an efficient *K-CAS*, as well as being highly portable, using only single word compare-and-swap instructions, with no memory reclaimer required. These results represent the first development in more than 10 years on the state of the art in the field.

References

1 D. Alistarh, W. Leiserson, A. Matveev, and N. Shavit. ThreadScan: Automatic and Scalable Memory Reclamation. In *SPAA*, 2015.

2 D. Alistarh, W. Leiserson, A. Matveev, and N. Shavit. Forkscan: Conservative Memory Reclamation for Modern Operating Systems. In *EuroSys*, 2017.

3 A. Amighi, S. Blom, and M. Huisman. VerCors: A Layered Approach to Practical Verification of Concurrent Software. In *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP*, 2016.

4 M. Arbel-Raviv and T. Brown. Reuse, Don't Recycle: Transforming Lock-Free Algorithms That Throw Away Descriptors. In *DISC*, 2017.

5 M. Batty. The C11 and C++11 Concurrency Model. `http://www.cl.cam.ac.uk/~mjb220/thesis/thesis.pdf`. Accessed: 2017-05-04.

6 T. Brown. A Template for Implementing Fast Lock-free Trees Using HTM. In *PODC*, 2017.

7 P. Celis. *Robin Hood Hashing*. PhD thesis, University of Waterloo, Waterloo, Ont., Canada, Canada, 1986.

8 C. Click. Lock-Free/Wait-Free Hash Table. `http://web.stanford.edu/class/ee380/Abstracts/070221_LockFreeHash.pdf`. Accessed: 2017-05-04.

9 N. Cohen and E. Petrank. Efficient memory management for lock-free data structures with optimistic access. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*, pages 254–263. ACM, 2015.

10 T. Cormen, C. Stein, R. Rivest, and C. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

11 U. Drepper. What Every Programmer Should Know About Memory, 2007.

12 J. Evans. JEMalloc, Retrieved 2018-08-06. Available at https://github.com/jemalloc/jemalloc.

13 S. Feldman, P. LaBorde, and D. Dechev. A Wait-Free Multi-Word Compare-and-Swap Operation. *International Journal of Parallel Programming*, 43(4):572–596, August 2015.

14 K. Fraser. Practical lock-freedom. Technical report, University of Cambridge, Computer Laboratory, 2004.

15 G. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures: In Pascal and C (2Nd Ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1991.

16 T. Harris. A Pragmatic Implementation of Non-blocking Linked-Lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC, 2001.

17 T. Harris, K. Fraser, and I. Pratt. A Practical Multi-word Compare-and-Swap Operation. In *Proceedings of the 16th International Conference on Distributed Computing*, DISC '02, pages 265–279, London, UK, UK, 2002. Springer-Verlag.

18 M. Herlihy, V. Luchangco, and M. Moir. Obstruction-Free Synchronization: Double-Ended Queues As an Example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCS '03. IEEE Computer Society, 2003.

19 M. Herlihy and J. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93. ACM, 1993.

20 M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, 1 edition, March 2008.

21 M. Herlihy, N. Shavit, and M. Tzafrir. Hopscotch Hashing. In *DISC '08: Proceedings of the 22nd international symposium on Distributed Computing*, pages 350–364, Berlin, Heidelberg, 2008. Springer-Verlag. `doi:10.1007/978-3-540-87779-0_24`.

22 M. Herlihy and J. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

23 R. Kelly. Source Code For Lock-Free Robin Hood Benchmark. `https://github.com/DaKellyFella/concurrent-robin-hood-hashing`. Accessed: 2018-11-14.

24 K. Rustan M. Leino and Peter Müller. *A Basis for Verifying Multi-threaded Programs*, pages 378–393. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

25 M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA. ACM, 2002.

26 M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, June 2004.

27 J. Nielsen and S. Karlsson. A Scalable Lock-free Hash Table with Open Addressing. *SIGPLAN Not.*, 51(8):33:1–33:2, February 2016.

28 D. Lea P. William, S. Adve. JSR 133: JavaTM Memory Model and Thread Specification Revision. `https://www.jcp.org/en/jsr/detail?id=133`. Accessed: 2017-05-04.

29 A. Padegs. System/370 extended architecture: Design considerations. *IBM Journal of Research and Development*, 27(3):198–205, 1983.

30 C. Purcell and T. Harris. *Non-blocking Hashtables with Open Addressing*, pages 108–121. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

31 R. Rajwar and J. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 294–305. IEEE Computer Society, 2001.

32 O. Shalev and N. Shavit. Split-Ordered Lists: Lock-Free Extensible Hash Tables. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2003.

33 D. Siakavaras, K. Nikas, G. Goumas, and N. Koziris. Massively Concurrent Red-Black Trees with Hardware Transactional Memory. In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, 2016.

34 H. Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3), 2005.

35 Rust Core Team. Rust Collections Hash Table. `https://doc.rust-lang.org/std/collections/struct.HashMap.html`. Accessed: 2017-05-07.

36 D. Terpstra, H. Jagode, H. You, and J. Dongarra. Collecting Performance Data with PAPI-C. In *Tools for High Performance Computing 2009*, pages 157–173, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

37 R. Yoo, C. Hughes, K. Lai, and R. Rajwar. Performance Evaluation of Intel; Transactional Synchronization Extensions for High-performance Computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC. ACM, 2013.

Parallel Combining: Benefits of Explicit Synchronization

Vitaly Aksenov[1]
ITMO University, Saint-Petersburg, Russia and Inria, Paris, France
aksenov@corp.ifmo.ru

Petr Kuznetsov
LTCI, Télécom ParisTech, Université Paris-Saclay, Paris, France
petr.kuznetsov@telecom-paristech.fr

Anatoly Shalyto
ITMO University, Saint-Petersburg, Russia
shalyto@mail.ifmo.ru

Abstract

A *parallel batched* data structure is designed to process synchronized *batches* of operations on the data structure using a parallel program. In this paper, we propose *parallel combining*, a technique that implements a *concurrent* data structure from a parallel batched one. The idea is that we explicitly synchronize concurrent operations into batches: one of the processes becomes a *combiner* which collects concurrent requests and initiates a parallel batched algorithm involving the owners (*clients*) of the collected requests. Intuitively, the cost of synchronizing the concurrent calls can be compensated by running the parallel batched algorithm.

We validate the intuition via two applications. First, we use parallel combining to design a concurrent data structure optimized for *read-dominated* workloads, taking a dynamic graph data structure as an example. Second, we use a novel parallel batched *priority queue* to build a concurrent one. In both cases, we obtain performance gains with respect to the state-of-the-art algorithms.

2012 ACM Subject Classification Computing methodologies → Concurrent computing methodologies, Computing methodologies → Parallel computing methodologies, Theory of computation → Distributed computing models, Theory of computation → Parallel computing models

Keywords and phrases concurrent data structure, parallel batched data structure, combining

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.11

Related Version The full version of the paper is available at [3], `http://arxiv.org/abs/1710.07588`.

1 Introduction

To ensure correctness of concurrent computations, various synchronization techniques are employed. Informally, synchronization is used to handle conflicts on *shared data*, e.g., resolving data races, or *shared resources*, e.g., allocating and deallocating memory. Intuitively, the more sophisticated conflict patterns a concurrent program is subject to – the higher are the incurred synchronization costs.

[1] This work was financially supported by the Government of Russian Federation (Grant 08-08)

© Vitaly Aksenov, Petr Kuznetsov, and Anatoly Shalyto;
licensed under Creative Commons License CC-BY
22nd International Conference on Principles of Distributed Systems (OPODIS 2018).
Editors: Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira; Article No. 11; pp. 11:1–11:16

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

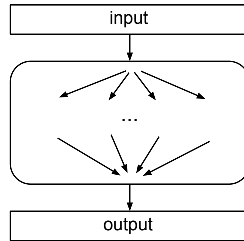


Figure 1 Execution of a parallel program.

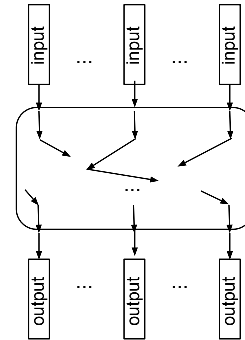


Figure 2 Execution on a concurrent data structure.

Let us consider a concurrent-software class which we call *parallel programs*. Provided an input, a parallel program aims at computing an output that satisfies a *specification*, i.e., an input-output relation (Figure 1). To boost performance, the program distributes the computation across multiple parallel processes. Parallel programs are typically written for two environments: for *static multithreading* and *dynamic multithreading* [11]. In *static multithreading*, each process is given its own program and these programs are written as a composition of *supersteps*. During a superstep, the processes perform conflict-free individual computations and, when done, synchronize to accumulate the results. In *dynamic multithreading*, the program is written using dynamically called *fork-join* mechanisms (or similar ones, e.g., `#pragma omp parallel` in OpenMP [6]). In both settings, synchronization only appears in a specific form: memory allocation/deallocation, and aggregating superstep computations or thread scheduling [17].

General-purpose *concurrent data structures*, such as stacks, binary search trees and priority queues, operate in a much less controlled environment. They are programmed to accept and process asynchronous operation calls, which come from multiple concurrent processes and may interleave arbitrarily. If we treat operation calls as inputs and their responses as outputs, we can say that inputs and outputs are distributed across the processes (Figure 2). It is typically expected that the interleaving operations match the high-level sequential semantics of the data type [24], which is hard to implement efficiently given diverse and complicated data-race patterns often observed in this kind of programs. Therefore, designing efficient and correct concurrent data structures requires a lot of ingenuity from the programmer. In particular, one should strive to provide the “just right” amount of synchronization. *Lock-based* data structures obviate data races by using fine-grained locking ensuring that contested data is accessed in a mutually exclusive way. *Wait-free* and *lock-free* data structures allow data races but mitigate their effects by additional mechanisms, such as *helping* where one process may perform some work on behalf of other processes [23].

As parallel programs are written for a restricted environment with simple synchronization patterns, they are typically easier to design than concurrent data structures. In this paper, we suggest benefiting from this complexity gap by building concurrent data structures *from* parallel programs. We describe a methodology of designing a concurrent data structure from its *parallel batched* counterpart [2]. A parallel batched data structure is a special case of a parallel program that accepts *batches* (sets) of operations on a given sequential data type and executes them in a parallel way. In our approach, we *explicitly* synchronize concurrent operations, assemble them into batches, and apply these batches on an *emulated* parallel batched data structure.

More precisely, concurrent processes share a set of *active requests* using any combining algorithm [31, 21, 15, 16]. One of the processes with an active request becomes a *combiner* and forms a *batch* from the requests in the set. Under the coordination of the combiner, the owners of the collected requests, called *clients*, apply the requests in the batch to the parallel batched data structure. As we show, this technique becomes handy when the overhead of *explicitly synchronizing* operation calls in batches is compensated by the advantages of involving clients into the computation using the parallel batched data structure.

We discuss two applications of parallel combining and experimentally validate performance gains. First, we design concurrent implementations optimized for *read-dominated* workloads given a sequential data structure. Intuitively, updates are performed sequentially and read-only operations are performed by the clients in parallel under the coordination of the combiner. In our performance analysis, we considered a *dynamic graph* data structure [25] that can be accessed for adding and removing edges (updates), as well as for checking the connectivity between pairs of vertices (read-only). Second, we apply parallel combining to *priority queue* that is subject to sequential bottlenecks for minimal-element extractions, while most insertions can be applied concurrently. As a side contribution, we propose a novel parallel batched priority queue, as no existing batched priority queue we are aware of can be efficiently used in our context. Our perfomance analysis shows that implementations based on parallel combining may outperform state-of-the-art algorithms.

Structure. The rest of the paper is organized as follows. In Section 2, we give preliminary definitions. In Section 3, we outline parallel combining technique. In Sections 4 and 5, we present applications of our technique. In Section 6, we report on the outcomes of our performance analysis. In Section 7, we overview the related work. We conclude in Section 8.

2 Background

Data types and data structures. A sequential *data type* is defined via a set of operations, a set of responses, a set of states, an initial state and a set of transitions. Each transition maps a state and an operation to a new state and a response. A *sequential implementation* (or *sequential data structure*) corresponding to a given data type specifies, for each operation, a sequential read-write algorithm, so that the specification of the data type is respected in every sequential execution.

We consider a system of n asynchronous *processes* (processors or threads of computation) that communicate by performing primitive operations on shared *base objects*. The primitive operations can be reads, writes, or *conditional* operations, such as test&set or compare&swap. A *concurrent implementation* (or *concurrent data structure*) of a given data type assigns, for each process and each operation of the data type, a state machine that is triggered whenever the process invokes an operation and specifies the sequence of *steps* (primitives on the base objects) the process needs to perform to complete the operation. We require the implementations to be *linearizable* with respect to the data type, i.e., we require that operations *take effect* instantaneously within their intervals [24].

Batched data structures. A *batched implementation* (or *batched data structure*) of a data type exports one function `apply`. This operation takes a *batch* (set) of data type operations as a parameter and returns responses for these operations that are consistent with some sequential application of the operations to the current state of the data structure, which is updated accordingly. We also consider extensions of the definition where we explicitly define

the "batched" data type via the *set* [30] or *interval* [10] linearizations. Such a data type takes a batch and a state, and returns a new state and a vector of responses.

For example, in the simplest form, a batched implementation may sequentially apply operations from a batch to the sequential data structure. But batched implementations may also use parallelism to accelerate the execution of the batch: we call these *parallel* batched implementations. We consider two types of parallel batched implementations: *static-multithreading* ones and *dynamic-multithreading* ones [11].

Static multithreading. A parallel batched data structure specifies a distinct (sequential) code to each process in PRAM-like models (PRAM [27], Bulk synchronous parallel model [37], Asynchronous PRAM [18], etc.). For example, in this paper, we provide a batched implementation of a priority queue in the Asynchronous PRAM model. The Asynchronous PRAM consists of n sequential processes, each with its own private local memory, communicating through the shared memory. Each process has its own program. Unlike the classical PRAM model, each process executes its instructions independently of the timing of the other processors. Each process performs one of the four types of instructions per tick of its local clock: global read, global write, local operation, or synchronization step. A synchronization step for a set S of processes is a logical point where each processor in S waits for all the processes in S to arrive before continuing its local program.

Dynamic multithreading. Here the parallel batched implementation is written as a sequential read-write algorithm using concurrency keywords specifiying logical parallelism, such as *fork*, *join* and *parallel-for* [11]. An execution of a batch can be presented as a directed acyclic graph (*DAG*) that unfolds dynamically. In the DAG, nodes represent unit-time sequential subcomputations, and edges represent control-flow dependencies between nodes. A node that corresponds to a "fork" has two or more outgoing edges and a node that corresponds to a "join" has two or more incoming edges. The batch is executed using a *scheduler* that chooses which DAG nodes to execute on each process. It can only execute *ready* nodes: not yet executed nodes whose predecessors have all been executed. The most commonly used *work-stealing* scheduler (e.g., [5]) operates as follows. Each process p is provided with a deque for ready nodes. When process p completes node u, it traverses successors of u and collects the ready ones. Then p selects one of the ready successors for execution and adds the remaining ready successors at the bottom of its deque. When p's deque is empty, it becomes a *thief*: it randomly picks a victim processor and steals from the top of the victim's deque.

3 Parallel Combining

In this section, we describe the *parallel combining* technique in a parameterized form: the parameters are specified depending on the application. We then discuss how to use the technique in transforming parallel batched programs into concurrent data structures.

3.1 Combining Data Structure

Our technique relies on a *combining* data structure $\mathbb{C}$ (e.g., the one used in [21]) that maintains a set of requests to a data structure and determines which process is a combiner. If the set of requests is not empty then exactly one process should be a combiner.

Elements stored in $\mathbb{C}$ are of `Request` type consisting of the following fields: 1) the method to be called and its input; 2) the response field; 3) the status of the request with a value in an application-specific `STATUS_SET`; 4) application-specific auxiliary fields. In our applications,

Listing 1 Parallel combining: pseudocode

```
1  Request:
2    method
3    input
4    res
5    status ∈ STATUS_SET
6    ...
7
8  execute(method, input):
9    req ← new Request()
10   req.method ← method
11   req.input ← input
12   req.status ← INITIAL
13   if ℂ.addRequest(req):
14     // combiner
15     A ← ℂ.getRequests()
16     COMBINER_CODE
17     ℂ.release()
18   else:
19     while req.status = INITIAL:
20       nop
21     CLIENT_CODE
22   return
```

STATUS_SET contains, by default, values INITIAL and FINISHED: INITIAL meaning that the request is in the initial state, and FINISHED meaning that the request is served.

$\mathbb{C}$ supports three operations: 1) addRequest(r : Request) inserts request r into the set, and the response indicates whether the calling process becomes a combiner or a client; 2) getRequests() returns a non-empty set of requests; and 3) release() is issued by the combiner to make $\mathbb{C}$ find another process to be a combiner.

In the following, we use any black-box implementation of $\mathbb{C}$ providing this functionality [31, 21, 15, 16].

3.2 Specifying Parameters

To perform an operation, a process executes the following steps (Listing 1): 1) it prepares a request, inserts the request into $\mathbb{C}$ using addRequest($\cdot$); 2) if the process becomes the combiner (i.e., addRequest($\cdot$) returned *true*), it collects requests from $\mathbb{C}$ using getRequests(), then it executes algorithm COMBINER_CODE, and, finally, it calls release() to enable another active process to become a combiner; 3) if the process is a client (i.e., addRequest($\cdot$) returned *false*), it waits until the status of the request becomes not INITIAL and, then, executes algorithm CLIENT_CODE.

To use our technique, one should therefore specify COMBINER_CODE, CLIENT_CODE, and appropriately modify Request type and STATUS_SET.

Note that *sequential combining* [31, 21, 16, 14] is a special case of parallel combining in which all the work is done by the combiner, and the client code is empty.

3.3 Parallel Batched Algorithms

We discuss how to build a concurrent data structure given a parallel batched one in one of two forms: for static or dynamic multithreading.

In the static multithreading case, each process is provided with a distinct version of `apply` function. We enrich `STATUS_SET` with `STARTED`. In `COMBINER_CODE`, the combiner collects the requests, sets their status to `STARTED`, performs the code of `apply` and waits for the clients to become `FINISHED`. In `CLIENT_CODE` the client waits until its request has `STARTED` status, performs the code of `apply` and sets the status of its request to `FINISHED`.

Suppose that we are given a parallel batched implementation for dynamic multithreading. One can turn it into a concurrent one using parallel combining with the *work-stealing* scheduler. Again, we enrich `STATUS_SET` with `STARTED`. In `COMBINER_CODE`, the combiner collects the requests and sets their status to `STARTED`. Then the combiner creates a working deque, puts there a new node of computational DAG with `apply` function and starts the work-stealing routine on processes-clients. Finally, the combiner waits for the clients to become `FINISHED`. In `CLIENT_CODE`, the client creates a working deque and starts the work-stealing routine.

In Section 5, we illustrate the use of parallel combining and parallel batched programs on the example of a priority queue.

4 Read-Optimized Concurrent Data Structures

Before discussing parallel batched algorithms, let us consider a natural application of parallel combining: data structures optimized for *read-dominated* workloads.

Suppose that we are given a sequential data structure D that supports *read-only* (not modifying the data structure) operations, the remaining operations are called *updates*. We assume a scenario where read-only operations dominate over other updates.

Now we explain how to set parameters of parallel combining for this application. At first, `STATUS_SET` consists of three elements `INITIAL`, `STARTED` and `FINISHED`. `Request` type does not have auxiliary fields.

In `COMBINER_CODE` (Listing 2 Lines 1-19), the combiner iterates through the set of collected requests A: if a request contains an update operation then the combiner executes it and sets its status to `FINISHED`; otherwise, the combiner adds the request to set R. Then the combiner sets the status of requests in R to `STARTED`. After that the combiner checks whether its own request is read-only. If so, it executes the method and sets the status of its request to `FINISHED`. Finally, the combiner waits until the status of the requests in R become `FINISHED`.

In `CLIENT_CODE` (Listing 2 Lines 21-24), the client checks whether its method is read-only. If so, the client executes the method and sets the status of the request to `FINISHED`.

▶ **Theorem 1.** *Algorithm in Listing 2 produces a linearizable concurrent data structure from a sequential one.*

Proof. Any execution of the algorithm can be viewed as a series of non-overlapping combining phases (Listing 2, Lines 2-19). We can group the operations into batches by the combining phase in which they are applied.

Each update operation is linearized at the point when the combiner applies this operation. Note that this is a correct linearization since all operations that are linearized before are already applied: the operations from preceding combining phases were applied during

Listing 2 Parallel combining in application to read-optimized data structures.

```
1  COMBINER_CODE:
2   R ← ∅
3
4   for r ∈ A:
5   if isUpdate(r.method):
6   apply(D, r.method, r.input)
7   r.status ← FINISHED
8   else:
9   R ← R ∪ r
10
11  for r ∈ R:
12  r.status ← STARTED
13  if req.status = STARTED:
14  apply(D, req.method, req.input)
15  req.status ← FINISHED
16
17  for r ∈ R:
18  while r.status = STARTED:
19  nop
20
21 CLIENT_CODE:
22  if not isUpdate(req.method):
23  apply(D, req.method, req.input)
24  req.status ← FINISHED
```

the preceding phases, while the operations from the current combining phase are applied sequentially by the combiner.

Each read-only operation is linearized at the point when the combiner sets the status of the corresponding request to STARTED. By the algorithm, a read-only operation observes all update operations that are applied before and during the current combining phase. Thus, the chosen linearization is correct. ◀

To evaluate the approach in practice we implement a concurrent *dynamic graph* data structure by Holm et al. and execute it in read-dominated environments [25] (Section 6.1).

5 Priority Queue

Priority queue is an abstract data type that maintains an ordered multiset and supports two operations:

- Insert(v) – inserts value v into the set;
- v ← ExtractMin() – extracts the smallest value from the set.

To the best of our knowledge, no prior parallel implementation of a priority-queue [32, 13, 9, 33] can be efficiently used in our context: their complexity inherently depends on the total number of processes in the system, regardless of the actual batch size. We therefore introduce a novel heap-based parallel batched priority-queue implementation in a form of COMBINER_CODE and CLIENT_CODE convenient for parallel combining. The concurrent priority queue is then derived from the described below parallel batched one using the approach presented in Section 3.3.

Here we give only the brief overview of our parallel batched algorithm. Please refer to the full version of the paper [3] for a detailed description.

In Section 6.2, we show that the resulting concurrent priority queue is able to outperform manually crafted state-of-the-art implementations.

5.1 Sequential Binary Heap

Our batched priority queue is based on the *sequential binary heap* by Gonnet and Munro [19], one of the simplest and fastest sequential priority queues. We briefly describe this algorithm below.

A binary heap of size m is represented as a complete binary tree with nodes indexed by $1, \ldots, m$. Each node v has at most two children: $2v$ and $2v+1$ (to exist, $2v$ and $2v+1$ should be less than or equal to m). For each node, the *heap property* should be satified: the value stored at the node is less than the values stored at its children.

The heap is represented with *size* m and an array a where $a[v]$ is the value at node v. Operations `ExtractMin` and `Insert` are performed as follows:

- `ExtractMin` records the value $a[1]$ as a response, copies $a[m]$ to $a[1]$, decrements m and performs the *sift down* procedure to restore the heap property. Starting from the root, for each node v on the path, we check whether value $a[v]$ is less than values $a[2v]$ and $a[2v+1]$. If so, then the heap property is satisfied and we stop the operation. Otherwise, we choose the child c, either $2v$ or $2v+1$, with the smallest value, swap values $a[v]$ and $a[c]$, and continue with c.
- `Insert(`x`)` initializes a variable val to x, increments m and traverses the path from the root to a new node m. For each node v on the path, if $val < a[v]$, then the two values are swapped. Then the operation continues with the child of v that lies on the path from v to node m. Reaching node m the operation sets its value to val.

The complexity is $O(\log m)$ steps per operation.

5.2 Setup

The heap is defined by its size m and an array a of Node objects. Node object has two fields: value val and boolean $locked$ (In the full version [3] it has an additional field).

`STATUS_SET` consists of three items: `INITIAL`, `SIFT` and `FINISHED`.

A Request object consists of: a method *method* to be called and its input argument v; a result *res* field; a *status* field and a node identifier *start*.

5.3 ExtractMin Phase

Combiner: ExtractMin preparation. The combiner withdraws requests A from combining data structure $\mathbb{C}$. It splits A into sets E and I: the set of ExtractMin requests and Insert requests. Then it finds $|E|$ nodes $v_1, \ldots, v_{|E|}$ of heap with the smallest values using the Dijkstra-like algorithm in $O(|E| \cdot \log |E|)$ steps: (i) create a heap of nodes ordered by values, put there the root 1; (ii) at each of the next $|E|$ steps withdraw the node v with the minimal value from the heap; (iii) put two children of v, $2v$ and $2v+1$, to the heap. The $|E|$ withdrawn nodes are the nodes with the $|E|$ minimal values. For each request $E[i]$, the combiner sets $E[i].res$ to $a[v_i].val$, $a[v_i].locked$ to `true`, and $E[i].start$ to v_i.

The combiner proceeds by pairing Insert requests in I with ExtractMin requests in E using the following procedure. Suppose that $\ell = \min(|E|, |I|)$. For each $i \in [1, \ell]$, the combiner sets $a[v_i].val$ to $I[i].v$ and $I[i].status$ to `FINISHED`, i.e., this Insert request becomes

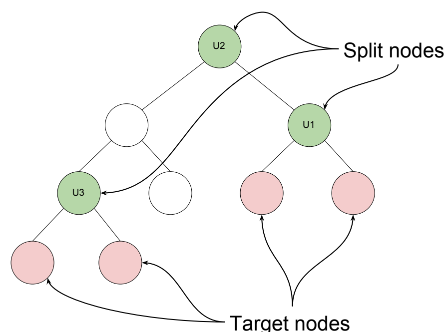


Figure 3 Split and target nodes.

completed. Then, for each $i \in [\ell + 1, |E|]$, the combiner sets $a[v_i].val$ to the value of the last node $a[m]$ and decrements m, as in the sequential algorithm. Finally, the combiner sets the status of all requests in E to SIFT.

Clients: ExtractMin phase. Briefly, the clients sift down the values in nodes $v_1, \ldots, v_{|E|}$ in parallel using hand-over-hand locking: the *locked* field of a node is set whenever there is a *sift down* operation working on that node.

A client c waits until the status of its request becomes SIFT. c starts sifting down from $req.start$. Suppose that c is currently at node v. c waits until the *locked* fields of the children become false. If $a[v].val$, the value of v, is less than the values in its children, then *sift down* is finished: c unsets $a[v].locked$ and sets the status of its request to FINISHED. Otherwise, let w be the child with the smallest value. Then c swaps $a[v].val$ and $a[w].val$, sets $a[w].locked$, unsets $a[v].locked$ and continues with node w.

If the request of the combiner is ExtractMin, it also runs the code above as a client. The combiner considers the ExtractMin phase completed when all requests in E have status FINISHED.

5.4 Insert Phase

For simplicity, we describe first the sequential algorithm.

At first, the combiner removes all completed requests from I. Then it initializes new nodes $m + 1, \ldots, m + |I|$ which we call *target nodes* and increments m by $|I|$. The nodes for which the subtrees of both children contain at least one target node are called *split nodes*. (See Figure 3 for an example of how target and split nodes can be defined.)

The combiner collects the values of the remaining Insert requests and sorts them: $r_1, \ldots, r_{|I|}$. Then it sets the status of these requests to FINISHED.

Now, we introduce InsertSet class: it consists of two sorted lists A and B. The combiner starts the following recursive procedure at the root with InsertSet s: $s.A$ contains $r_1, \ldots, r_{|I|}$ while $s.B$ is empty. Suppose that the procedure is called on node v and InsertSet s. Let min be the minimum out of the first element of $s.A$ and the first element of $s.B$. If v is a target node then the combiner sets $a[v].res$ to min and withdraws m from the corresponding list. Otherwise, the combiner compares $a[v].res$ with min: if $a[v].res$ is smaller, then it does nothing; otherwise, it appends $a[v].res$ to the end of $s.B$ (note that $s.B$ remains sorted because $s.B$ consists only of values that were ancestors in the heap), withdraws min from the corresponding list and sets $a[v].res$ to min.

If v is not the split node the combiner calls the recursive procedure on the child with target nodes in the subtree and with InsertSet s. Otherwise, the combiner calculates inL and inR – the number of target nodes in the left and right subtrees of v. Suppose, for simplicity, that inL is less than inR (the opposite case can be resolved similarly). The combiner splits s into two parts: create InsertSet s_L, move $\min(inL, |s.A|)$ first values from $s.A$ to $s_L.A$ and move $\min(inL - |s_L.A|, |s.B|)$ first values from $s.B$ to $s_L.B$. Finally, it calls the recursive procedure on the left child with InsertSet s_L and on the right child with InsertSet s.

This algorithm works in $O(\log m + c \log c)$ steps (and can be optimized to $O(\log m + c)$ steps), where m is the size of the queue and c is the number of Insert requests to apply. Note that this algorithm is almost non-parallelizable due to its small complexity, and our parallel algorithm is only developed to reduce constant factors.

Now, we construct a parallel algorithm for the Insert phase. We enrich Node object with the IntegerSet field *split*. The combiner sets the *start* field of the first client ($i[1].start$) to the root 1, while *start* fields of other clients to the right children of split nodes (we have exactly $|I| - 1$ split nodes). Then it intializes the *split* field of the root as the IntegerSet s is initialized at the beginning of the sequential algorithm: list A contains values of requests while list B is empty.

Each client waits until the *split* field of the corresponding *start* node is non-null. Then it reads this IntegerSet: the values from this set should be inserted in the subtree. Finally, the client performs the procedure similar to the recursive procedure from the sequential algorithm except for one difference: when it reaches a split node instead of going recursively to left and right children, it splits InsertSet to s_L and s_R of sizes inL and inR, puts s_R into the *split* field of the right child (in order to wake another client) and continues with the left child and s_L.

For further details about the parallel algorithm we refer to the full version of the paper [3].

6 Experiments

We evaluate Java implementations of our data structures on a 4-processor AMD Opteron 6378 2.4 GHz server with 16 threads per processor (yielding 64 threads in total), 512 Gb of RAM, running Ubuntu 14.04.5 with Java 1.8.0_111-b14 and HotSpot JVM 25.111-b14.

6.1 Concurrent Dynamic Graph

To illustrate how parallel combining can be used to construct read-optimized concurrent data structures, we took the sequential dynamic graph implementation by Holm et al. [25]. This data structure supports two update methods: an insertion of an edge and a deletion of an edge; and one read-only method: a connectivity query that tests whether two vertices are connected.

We compare our implementation based on parallel combining (PC) with *flat combining* [21] as a combining data structure against three others: (1) Lock, based on ReentrantLock from `java.util.concurrent`; (2) RW Lock, based on ReentrantReadWriteLock from `java.util.concurrent`; and (3) FC, based on *flat combining* [21]. The code is available at `https://github.com/Aksenov239/concurrent-graph`.

We consider workloads parametrized with: 1) the fraction x of connectivity queries (50%, 80% or 100%, as we consider read-dominated workloads); 2) the set of edges E: edges of a single random tree, or edges of ten random trees; 3) the number of processes P (from 1 to 64). We prepopulate the graph on 10^5 vertices with edges from E: we insert each edge with probability $\frac{1}{2}$. Then we start P processes. Each process repeatedly performs operations:

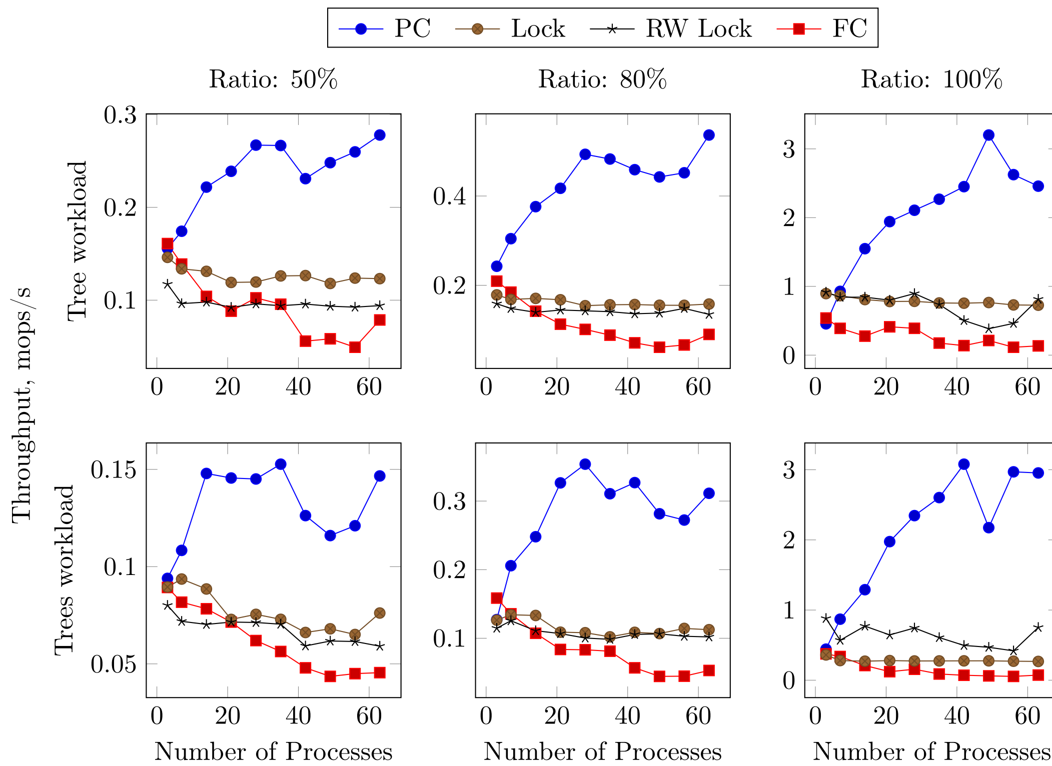


Figure 4 Dynamic graph implementations.

1) with probability x, it calls a connectivity query on two vertices chosen uniformly at random; 2) with probability $1 - \frac{x}{2}$, it inserts an edge chosen uniformly at random from E; 3) with probability $1 - \frac{x}{2}$, it deletes an edge chosen uniformly at random from E.

We denote the workloads with E as a single tree as *Tree* workloads, and other workloads as *Trees* workloads. *Tree* workloads are interesting because they show the degenerate case: the dynamic graph behaves as a dynamic tree. In this case, about 50% of update operations successfully change the spanning fprest, while other update operations only check the existence of the edge and do not modify the graph. *Trees* workloads are interesting because a reasonably small number (approximately, 5-10%) of update operations modify the maintained set of all edges and the underlying complex data structure that maintains a spanning forest (giving in total the squared logarithmic complexity), while other update operations can only modify the set of edges but cannot modify the underlying complex data structure (giving in total the logarithmic complexity).

For each setting and each algorithm, we run the corresponding workload for 10 seconds to warmup HotSpot JVM and then we run the workload five more times for 10 seconds. The average throughput of the last five runs is reported in Figure 4.

From the plots we can infer two general observations: PC exhibits the highest thoughput over all considered implementations and it is the only one whose throughput scales up with the number of the processes. On the 100% workload we expect the throughput curve to be almost linear since all operations are read-only and can run in parallel. The plots almost confirm our expectation: the curve of the throughput is a linear function with coefficient $\frac{1}{2}$ (instead of the ideal coefficient 1). We note that this is almost the best we can achieve: a combiner typically collects operations of only approximately half the number of working

processes. In addition, the induced overhead is still perceptible, since each connectivity query works in just logarithmic time. With the decrease of the fraction of read-only operations we expect that the throughput curve becomes flatter, as plots for the 50% and 80% workloads confirm.

It is also interesting to point out several features of other implementations. At first, FC implementation works slightly worse than Lock and RW Lock. This might be explained as follows. Lock implementations (`ReentrantLock` and `ReentrantReadWriteLock`) behind Lock and RWLock implementations are based on CLH Lock [12] organized as a *queue*: every competing process is appended to the queue and then waits until the previous one releases the lock. Operations on the dynamic graph take significant amount of time, so under high load when the process finishes its operation it appends itself to the queue in the lock without any contention. Indeed, all other processes are likely to be in the queue and, thus, no process can contend. By that the operations by processes are serialized with almost no overhead. In contrast, the combining procedure in FC introduces non-negligible overhead related to gathering the requests and writing them into requests structures.

Second, it is interesting to observe that, against the intuition, RWLock is not so superior with respect to Lock on read-only workloads. As can be seen, when there are update operations in the workload RWLock works even worse than Lock. We relate this to the fact that the overhead hidden inside `ReentrantReadWriteLock` spent on manipulation with read and write requests is bigger than the overhead spent by `ReentrantLock`. With the increase of the percentage of read-only operations the difference between Lock and RWLock diminishes and RWLock becomes dominant since read operations become more likely to be applied concurrently (for example, on 50% it is normal to have an execution without any parallelization: read operation, write operation, read operation, and so on). However, on 100% one could expect that RWLock should exhibit ideal throughput. Unfortunately, in this case, under the hood `ReentrantReadWriteLock` uses compare&swap on the shared variable that represents the number of current read operations. Read-only operations take enough time but not enough to amortize the considerable traffic introduced by concurrent compare&swaps. Thus, the plot for RWLock is almost flat, getting even slightly worse with the increase of the number of processes, and we blame the traffic for this.

6.2 Priority Queue

We run our algorithm (PC) with *flat combining* [21] as a combining data structure against six state-of-the-art concurrent priority queues: (1) the lock-free skip-list by Linden and Johnson (Linden SL [28]), (2) the lazy lock-based skip-list (Lazy SL [23]), (3) the non-linearizable lock-free skip-list by Herlihy and Shavit (SkipQueue [23]) as an adaptation of Lotan and Shavit's algorithm [34], (4) the lock-free skip-list from Java library (JavaLib), (5) the binary heap with flat combining (FC Binary [21]), and (6) the pairing heap with flat combining (FC Pairing [21]). [2] The code is available at `https://github.com/Aksenov239/FC-heap`.

We consider workloads parametrized by: 1) the initial size of the queue S ($8 \cdot 10^5$ or $8 \cdot 10^6$); and 2) the number P of working processes (from 1 to 64). We prepopulate the queue with S random integers chosen uniformly from the range $[0, 2^{31} - 1]$. Then we start P processes, and each process repeatedly performs operations: with equal probability it either inserts a random value taken uniformly from $[0, 2^{31} - 1]$ or extracts the minimum value.

[2] We are aware of the cache-friendly priority queue by Braginsky et al. [8], but we do not have its Java implementation.

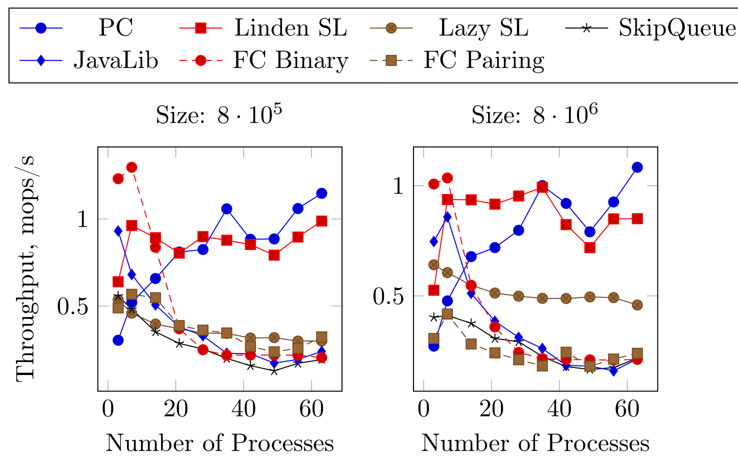


Figure 5 Priority Queue implementations.

For each setting and each algorithm, we run the corresponding workload for 10 seconds to warmup HotSpot JVM and then we run the workload five more times for 10 seconds. The average throughput of the last five runs is reported in Figure 5.

On a small number of processes (< 15), PC performs worse than other algorithms. With respect to Linden SL, Lazy SL, SkipQueue and JavaLib this can be explained by two different issues:

- Synchronization incurred by PC is not compensated by the work done;
- Typically, a combiner collects operations of only approximately half the processes, thus, we "overserialize", i.e., only $\frac{n}{2}$ operations can be performed in parallel.

In contrast, on small number of processes, the other four algorithms can perform operations almost with no contention. With respect to algorithms based on flat combining, FC Binary and FC Pairing, our algorithm is simply slower on one process than the simplest sequential binary and pairing heap algorithms.

With the increase of the number of processes the synchronization overhead significantly increases for all algorithms (in addition to the fact that FC Binary and FC Pairing cannot scale). As a result, starting from 15 processes, PC outperforms all algorithms except for Linden SL. Linden SL relaxes the contention during ExtractMin operations, and it helps to keep the throughput approximately constant. At approximately 40 processes the benefits of the parallel batched algorithm in PC starts prevailing the costs of explicit synchronization, and our algorithms overtakes Linden SL.

It is interesting to note that FC Binary performs very well when the number of processes is small: the overhead on the synchronization is very small, the processes are from the same core and the simplest binary heap performs operations very fast.

7 Related Work

To the best of our knowledge, Yew et al. [38] were the first to propose combining concurrent operations. They introduced a *combining tree*: processes start at distinct leaves, traverse upwards, and gain exclusive access by reaching the root. If, during the traversal, two processes access the same tree node, one of them adopts the operations of another and continues the traversal, while the other waits until its operations are completed. Several improvements of this technique have been discussed, such as adaptive combining tree [36], barrier implementations [20, 29] and counting networks [35].

A different approach was proposed by Oyama et al. [31]. Here the data structure is protected by a lock. A thread with a new operation to be performed adds it to a list of submitted requests and then tries to acquire the lock. The winner of the lock performs the pending requests on behalf of other processes from the list in LIFO order. The main drawback of this approach is that all processes have to perform CAS on the head of the list. The *flat combining* technique presented by Hendler et al. [21] addresses this issue by replacing the list of requests with a *publication list* which maintains a distinct *publication record* per participating process. A process puts its new operation in its publication record, and the publication record is only maintained in the list if the process is "active enough". This way the processes generally do not contend on the head of the list. Variations of flat combining were later proposed for various contexts [15, 16, 26, 14].

Hierarchical combining [22] is the first attempt to improve performance of combining using the computational power of clients. The list of requests is split into blocks, and each of these blocks has its own combiner. The combiners push the combined requests from the block into the second layer implemented as the standard flat combining with one combiner. This approach, however, may be sub-optimal as it does not involve *all* clients. Moreover, this approach works only for specific data structures, such as stacks or unfair synchronous queues, where operations could be combined without accessing the data structure.

In a different context, Agrawal et al. [2] suggested to use a *parallel batched* data structure instead of a concurrent one. They provide provable bounds on the running time of a dynamic multithreaded parallel program using P processes and a specified *scheduler*. The proposed scheduler extends the work-stealing scheduler by maintaining separate *batch* work-stealing deques that are accessed whenever processes have operations to be performed on the abstract data type. A process with a task to be performed on the data structure stores it in a *request array* and tries to acquire a global lock. If succeeded, the process puts the task to perform the batch update in its batch deque. Then all the processes with requests in the request array run the work-stealing routine on the batch deques until there are no tasks left. The idea of [2] is similar to ours. However, their algorithm is designed for systems with the fixed set of processes, whereas we allow the processes to join and leave the execution. From a more formal perspective, our goals are different: we aim at improving the performance of a *concurrent data structure* while their goal was to establish bounds on the running time of a *parallel program in dynamic multithreading*. Furthemore, implementing a concurrent data structure from its parallel batched counterpart for dynamic multithreading is only one of the applications of our technique, as sketched in Section 3.3.

8 Concluding remarks

Besides performance gains, parallel combining can potentially bring other interesting benefits.

First, a parallel batched implementation is typically provided with bounds on the running time. The use of parallel combining might allow us to derive bounds on the operations of resulting *concurrent* data structures. Consider, for example, a binary search tree. To balance the tree, state-of-the-art concurrent algorithms use the relaxed AVL-scheme [7]. This scheme guarantees that the height of the tree never exceeds the contention level (the number of concurrent operations) plus the logarithm of the tree size. Applying parallel combining to a parallel batched binary search tree (e.g., [4]), we get a concurrent tree with a strict logarithmic bound on the height.

Second, the technique might enable the first ever concurrent implementation of certain data types, for example, a *dynamic tree* [1].

As shown in Section 6, our concurrent priority queue performs well compared to state-of-the-art algorithms. A possible explanation is that the underlying parallel batched implementation is designed for *static* multithreading and, thus, it has little synchronization overhead. This might not be the case for implementations based on *dynamic* multithreading, where the overhead induced by the scheduler can be much higher. We intend to explore this distinction in the forthcoming work.

References

1. Umut A. Acar, Vitaly Aksenov, and Sam Westrick. Brief Announcement: Parallel Dynamic Tree Contraction via Self-Adjusting Computation. In *SPAA*, pages 275–277. ACM, 2017.
2. Kunal Agrawal, Jeremy T Fineman, Kefu Lu, Brendan Sheridan, Jim Sukha, and Robert Utterback. Provably good scheduling for parallel programs that use data structures through implicit batching. In *SPAA*, pages 84–95. ACM, 2014.
3. Vitaly Aksenov, Petr Kuznetsov, and Anatoly Shalyto. Parallel Combining: Benefits of Explicit Synchronization. *CoRR*, abs/1710.07588, 2018. `arXiv:1710.07588`.
4. Guy E Blelloch, Daniel Ferizovic, and Yihan Sun. Just join for parallel ordered sets. In *SPAA*, pages 253–264. ACM, 2016.
5. Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.
6. OpenMP Architecture Review Board. OpenMP Application Interface. `http://www.openmp.org`, 2008.
7. Luc Bougé, Joaquim Gabarro, Xavier Messeguer, Nicolas Schabanel, et al. Height-relaxed AVL rebalancing: A unified, fine-grained approach to concurrent dictionaries. Technical report, ENS Lyon, 1998.
8. Anastasia Braginsky, Nachshon Cohen, and Erez Petrank. CBPQ: High performance lock-free priority queue. In *Euro-Par*, pages 460–474. Springer, 2016.
9. Gerth Stølting Brodal, Jesper Larsson Träff, and Christos D Zaroliagis. A parallel priority queue with constant time operations. *Journal of Parallel and Distributed Computing*, 49(1):4–21, 1998.
10. Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. Specifying Concurrent Problems: Beyond Linearizability and up to Tasks. In *DISC*, pages 420–435, 2015.
11. Thomas H Cormen, Charles Eric Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. The MIT press, 3rd edition, 2009.
12. Travis Craig. Building FIFO and priorityqueuing spin locks from atomic swap. Technical report, Technical Report TR 93-02-02, University of Washington, 02 1993., 1993.
13. Narsingh Deo and Sushil Prasad. Parallel heap: An optimal parallel priority queue. *The Journal of Supercomputing*, 6(1):87–98, 1992.
14. Dana Drachsler-Cohen and Erez Petrank. LCD: Local Combining on Demand. In *OPODIS*, pages 355–371. Springer, 2014.
15. Panagiota Fatourou and Nikolaos D Kallimanis. A highly-efficient wait-free universal construction. In *SPAA*, pages 325–334. ACM, 2011.
16. Panagiota Fatourou and Nikolaos D Kallimanis. Revisiting the combining synchronization technique. In *ACM SIGPLAN Notices*, volume 47, pages 257–266. ACM, 2012.
17. Matteo Frigo, Charles E Leiserson, and Keith H Randall. The implementation of the Cilk-5 multithreaded language. *ACM Sigplan Notices*, 33(5):212–223, 1998.
18. Phillip B Gibbons. A more practical PRAM model. In *SPAA*, pages 158–168. ACM, 1989.
19. Gaston H Gonnet and J Ian Munro. Heaps on heaps. *SIAM Journal on Computing*, 15(4):964–971, 1986.

20 Rajiv Gupta and Charles R Hill. A scalable implementation of barrier synchronization using an adaptive combining tree. *International Journal of Parallel Programming*, 18(3):161–180, 1989.

21 Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *SPAA*, pages 355–364, 2010.

22 Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Scalable Flat-Combining Based Synchronous Queues. In *DISC*, pages 79–93. Springer, 2010.

23 Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming.* Morgan Kaufmann, 2012.

24 Maurice Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

25 Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM (JACM)*, 48(4):723–760, 2001.

26 Brandon Holt, Jacob Nelson, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Flat combining synchronized global data structures. In *7th International Conference on PGAS Programming Models*, page 76, 2013.

27 Joseph JáJá. *An introduction to parallel algorithms*, volume 17. Addison-Wesley Reading, 1992.

28 Jonatan Lindén and Bengt Jonsson. A skiplist-based concurrent priority queue with minimal memory contention. In *International Conference On Principles Of Distributed Systems*, pages 206–220. Springer, 2013.

29 John M Mellor-Crummey and Michael L Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

30 Gil Neiger. Set-Linearizability. In *PODC*, page 396, 1994.

31 Yoshihiro Oyama, Kenjiro Taura, and Akinori Yonezawa. Executing parallel programs with synchronization bottlenecks efficiently. In *Proceedings of the International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications*, volume 16, 1999.

32 Maria Cristina Pinotti and Geppino Pucci. Parallel priority queues. *Information Processing Letters*, 40(1):33–40, 1991.

33 Peter Sanders. Randomized priority queues for fast parallel access. *Journal of Parallel and Distributed Computing*, 49(1):86–97, 1998.

34 Nir Shavit and Itay Lotan. Skiplist-based concurrent priority queues. In *IPDPS*, pages 263–268. IEEE, 2000.

35 Nir Shavit and Asaph Zemach. Diffracting trees. *ACM Transactions on Computer Systems (TOCS)*, 14(4):385–428, 1996.

36 Nir Shavit and Asaph Zemach. Combining funnels: a dynamic approach to software combining. *Journal of Parallel and Distributed Computing*, 60(11):1355–1387, 2000.

37 Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

38 Pen-Chung Y, Nian-Feng T, et al. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Transactions on Computers*, 100(4):388–395, 1987.

Specification and Implementation of Replicated List: The Jupiter Protocol Revisited

Hengfeng Wei
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
hfwei@nju.edu.cn

Yu Huang[1]
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
yuhuang@nju.edu.cn

Jian Lu
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
lj@nju.edu.cn

Abstract

The replicated list object is frequently used to model the core functionality of replicated collaborative text editing systems. Since 1989, the convergence property has been a common specification of a replicated list object. Recently, Attiya et al. proposed the strong/weak list specification and conjectured that the well-known Jupiter protocol satisfies the weak list specification. The major obstacle to proving this conjecture is the mismatch between the global property on all replica states prescribed by the specification and the local view each replica maintains in Jupiter using data structures like 1D buffer or 2D state space. To address this issue, we propose CJupiter (Compact Jupiter) based on a novel data structure called n-ary ordered state space for a replicated client/server system with n clients. At a high level, CJupiter maintains only a single n-ary ordered state space which encompasses exactly all states of each replica. We prove that CJupiter and Jupiter are equivalent and that CJupiter satisfies the weak list specification, thus solving the conjecture above.

2012 ACM Subject Classification Computing methodologies → Distributed computing methodologies, Software and its engineering → Correctness, Human-centered computing → Collaborative and social computing systems and tools

Keywords and phrases Collaborative text editing systems, Replicated list, Concurrency control, Strong/weak list specification, Operational transformation, Jupiter protocol

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.12

Related Version A full version is available at [25], `https://arxiv.org/abs/1708.04754`.

Funding This work is supported by the National 973 Program of China (No. 2015CB352202) and the National Natural Science Foundation of China (No. 61690204, 61702253).

1 Introduction

Collaborative text editing systems, like Google Docs [2], Apache Wave [1], or wikis [11], allows multiple users to concurrently edit the same document. For availability, such systems often replicate the document at several *replicas*. For low latency, replicas are required to

[1] Contact Author.

© Hengfeng Wei, Yu Huang, and Jian Lu;
licensed under Creative Commons License CC-BY
22nd International Conference on Principles of Distributed Systems (OPODIS 2018).
Editors: Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira; Article No. 12; pp. 12:1–12:16

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

respond to user operations immediately without any communication with others and updates are propagated asynchronously.

The *replicated list object* has been frequently used to model the core functionality (e.g., insertion and deletion) of replicated collaborative text editing systems [8, 13, 26, 5]. A common specification of a replicated list object is the *convergence* property, proposed by Ellis et al. [8]. It requires the *final lists at all replicas* be identical after executing the same set of user operations. Recently, Attiya et al. [5] proposed the strong/weak list specification. Beyond the convergence property, the strong/weak list specification specifies global properties on *intermediate states* going through by replicas. Attiya et al. [5] have proved that the existing RGA protocol [16] satisfies the strong list specification. Meanwhile, it is *conjectured* that the well-known Jupiter protocol [13, 26], which is behind Google Docs [3] and Apache Wave [4], satisfies the weak list specification.

Jupiter adopts a *centralized server* replica for propagating updates [2], and client replicas are connected to the server replica via FIFO channels; see Figure 1 [3]. Jupiter relies on the technique of operational transformations (OT) [8, 20] to achieve convergence. The basic idea of OT is for each replica to execute any local operation immediately and to transform a remote operation so that it takes into account the concurrent operations previously executed at the replica. Consider a replicated list system consisting of replicas R_1 and R_2 which initially hold the same list (Figure 2). Suppose that user 1 invokes $o_1 = \textsc{Ins}(f, 1)$ at R_1 and concurrently user 2 invokes $o_2 = \textsc{Del}(5)$ at R_2. After being executed locally, each operation is sent to the other replica. Without OT (Figure 2a), the states of two replicas diverge. With the OT of o_1 and o_2 (Figure 2b), o_2 is transformed to $o_2' = \textsc{Del}(6)$ at R_1, taking into account the fact that o_1 has inserted an element at position 1. Meanwhile, o_1 remains unchanged. As a result, two replicas converge to the same list. We note that although the idea of OT is straightforward, many OT-based protocols for replicated list are hard to understand and some of them have even been shown incorrect with respect to convergence [8, 20, 22].

The major obstacle to proving that Jupiter satisfies the weak list specification is the *mismatch* between the *global property* on all states prescribed by such a specification and the *local view* each replica maintains in the protocol. On the one hand, the weak list specification requires that states across the system are pairwise compatible [5]. That is, for any pair of (list) states, there cannot be two elements a and b such that a precedes b in one state but b precedes a in the other. On the other hand, Jupiter uses data structures like 1D buffer [18] or 2D state space [13, 26] which are not "compact" enough to capture all replica states in one. In particular, Jupiter maintains $2n$ 2D state spaces for a system with n clients [26]: Each client maintains a single state space which is synchronized with those of other clients via its counterpart state space maintained by the server. Each 2D state space of a client (as well as its counterpart at the server) consists of a local dimension and a global dimension, keeping track of the operations processed by the client itself and the others, respectively. In this way, replica states of Jupiter are dispersed in multiple 2D state spaces maintained locally at individual replicas.

To resolve the mismatch, we propose CJupiter (Compact Jupiter), a variant of Jupiter, which uses a novel data structure called *n-ary ordered state space* for a system with n clients. CJupiter is compact in the sense that at a high level, it maintains only a single n-ary ordered state space which encompasses exactly all states of each replica. Each replica behavior

[2] Since replicas are required to respond to user operations immediately, the client/server architecture does not imply that clients process operations in the same order.
[3] The details about Figure 1 will be described in Examples 4 and 13.

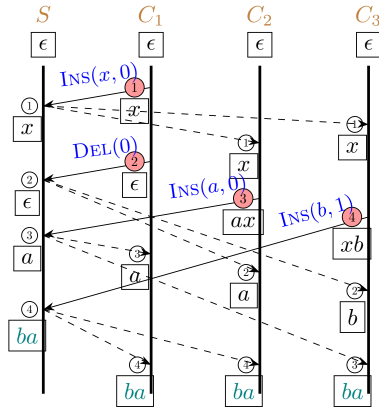
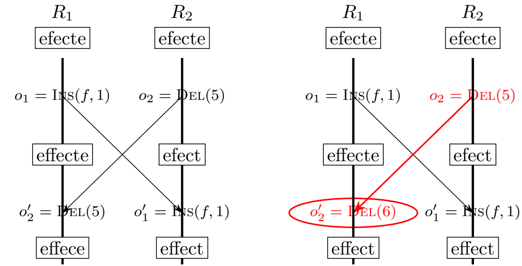


Figure 1 A schedule of four operations adapted from [5], involving a server replica s and three client replicas c_1, c_2, and c_3. The circled numbers indicate the order in which the operations are received at the server. The list contents produced by CJupiter (Section 3) are shown in boxes.



(a) Without OT, the states of R_1 and R_2 diverge.

(b) With OT, R_1 and R_2 converge to the same state.

Figure 2 Illustrations of OT (adapted from [9]).

corresponds to a path going through this state space. This makes it feasible for us to reason about global properties and finally prove that Jupiter satisfies the weak list specification, thus solving the conjecture of Attiya et al. The roadmap is as follows:

- (Section 3) We propose CJupiter based on the n-ary ordered state space data structure.
- (Section 4) We prove that CJupiter is equivalent to Jupiter in the sense that the behaviors of corresponding replicas of these two protocols are the same under the same schedule of operations. Jupiter is slightly optimized in implementation at clients (but not at the server) by eliminating redundant OTs, which, however, has obscured the similarities among clients and led to the mismatch discussed above.
- (Section 5) We prove that CJupiter satisfies the weak list specification. Thanks to the "compactness" of CJupiter, we are able to focus on a single n-ary ordered state space which provides a global view of all possible replica states.

Section 2 presents preliminaries on specifying replicated list data type and OT. Section 6 describes related work. Section 7 concludes the paper. The full paper [25] contains proofs and pseudocode.

2 Preliminaries: Replicated List and Operational Transformation

We describe the system model and specifications of replicated list in the framework for specifying replicated data types [7, 6, 5].

2.1 System Model

A highly-available replicated data store consists of *replicas* that process user operations on the replicated objects and communicate updates to each other with messages. To be *highly-available*, replicas are required to respond to user operations immediately without any communication with others. A *replica* is defined as a state machine $R = (\Sigma, \sigma_0, E, \Delta)$, where *1)* Σ is a set of states; *2)* $\sigma_0 \in \Sigma$ is the initial state; *3)* E is a set of possible events; and *4)* $\Delta : \Sigma \times E \to \Sigma$ is a transition function. The state transitions determined by Δ are local

steps of a replica, describing how it interacts with the following three kinds of *events* from users and other replicas:

- do(o, v): a user invokes an operation $o \in \mathcal{O}$ on the replicated object and immediately receives a response $v \in \mathit{Val}$. We leave the users unspecified and say that the replica *generates* the operation o;
- send(m): the replica sends a message m to some replicas; and
- receive(m): the replica receives a message m.

A *protocol* is a collection $\mathcal{R}$ of replicas. An *execution* α of a protocol $\mathcal{R}$ is a sequence of all events occurring at the replicas in $\mathcal{R}$. We denote by $R(e)$ the replica at which an event e occurs. For an execution (or generally, an event sequence) α, we denote by $e \prec_\alpha e'$ (or $e \prec e'$) that e precedes e' in α. An execution α is *well-formed* if for every replica R: *1)* the subsequence of events $\langle e_1, e_2, \ldots \rangle$ at R, denoted $\alpha|_R$, is well-formed, namely there is a sequence of states $\langle \sigma_1, \sigma_2, \ldots \rangle$, such that $\sigma_i = \Delta(\sigma_{i-1}, e_i)$ for all i; and *2)* every receive(m) event at R is preceded by a send(m) event in α. We consider only *well-formed* executions.

We are often concerned with replica behaviors and states when studying a protocol. The *behavior* of replica R in α is a sequence of the form: $\sigma_0, e_1, \sigma_1, e_2, \ldots$, where $\langle e_1, e_2, \ldots \rangle = \alpha|_R$ and $\sigma_i = \Delta(\sigma_{i-1}, e_i)$ for all i. A *replica state* σ of R in α can be represented by the events in a prefix of $\alpha|_R$ it has processed. Specifically, $\sigma_0 = \langle\rangle$ and $\sigma_i = \sigma_{i-1} \circ e_i = \langle e_1, e_2, \ldots, e_i \rangle$.

We now define the causally-before, concurrent, and totally-before relations on events in an execution. When restricted to the do events only, they define relations on user operations. In an execution α, event e is *causally before* e', denoted $e \xrightarrow{\text{hb}_\alpha} e'$ (or $e \xrightarrow{\text{hb}} e'$), if one of the following conditions holds [10]: *1)* Thread of execution: $R(e) = R(e') \wedge e \prec_\alpha e'$; *2)* Message delivery: $e = \text{send}(m) \wedge e' = \text{receive}(m)$; *3)* Transitivity: $\exists e'' \in \alpha : e \xrightarrow{\text{hb}_\alpha} e'' \wedge e'' \xrightarrow{\text{hb}_\alpha} e'$. Events $e, e' \in \alpha$ are *concurrent*, denoted $e \parallel_\alpha e'$ (or $e \parallel e'$), if it is neither $e \xrightarrow{\text{hb}_\alpha} e'$ nor $e' \xrightarrow{\text{hb}_\alpha} e$. A relation on events in an execution α, denoted $e \xrightarrow{\text{tb}_\alpha} e'$ (or $e \xrightarrow{\text{tb}} e'$), is a *totally-before* relation *consistent with* the causally-before relation '$\xrightarrow{\text{hb}_\alpha}$' on events in α if it is total: $\forall e, e' \in \alpha : e \xrightarrow{\text{tb}_\alpha} e' \vee e' \xrightarrow{\text{tb}_\alpha} e$, and it is consistent: $\forall e, e' \in \alpha : e \xrightarrow{\text{hb}_\alpha} e' \implies e \xrightarrow{\text{tb}_\alpha} e'$.

2.2 Specifying Replicated Objects

A replicated object is specified by a set of abstract executions which record user operations (corresponding to do events) and visibility relations on them [7]. An *abstract execution* is a pair $A = (H, \text{vis})$, where H is a sequence of do events and $\text{vis} \subseteq H \times H$ is an acyclic *visibility* relation such that *1)* if $e_1 \prec_H e_2$ and $R(e_1) = R(e_2)$, then $e_1 \xrightarrow{\text{vis}} e_2$; *2)* if $e_1 \xrightarrow{\text{vis}} e_2$, then $e_1 \prec_H e_2$; and *3)* vis is transitive: $(e_1 \xrightarrow{\text{vis}} e_2 \wedge e_2 \xrightarrow{\text{vis}} e_3) \implies e_1 \xrightarrow{\text{vis}} e_3$.

An abstract execution $A' = (H', \text{vis}')$ is a *prefix* of another abstract execution $A = (H, \text{vis})$ if H' is a prefix of H and $\text{vis}' = \text{vis} \cap (H' \times H')$. A *specification* $\mathcal{S}$ of a replicated object is a *prefix-closed* set of abstract executions, namely if $A \in \mathcal{S}$, then $A' \in \mathcal{S}$ for each prefix A' of A. A protocol $\mathcal{R}$ *satisfies* a specification $\mathcal{S}$, denoted $\mathcal{R} \models \mathcal{S}$, if any (concrete) execution α of $\mathcal{R}$ *complies with* some abstract execution $A = (H, \text{vis})$ in $\mathcal{S}$, namely $\forall R \in \mathcal{R} : H|_R = \alpha|_R^{\text{do}}$, where $\alpha|_R^{\text{do}}$ is the subsequence of do events of replica R in α.

2.3 Replicated List Specification

A replicated list object supports three types of user operations [5] (U for some universe):

- $\textsc{Ins}(a, p)$: inserts $a \in U$ at position $p \in \mathbb{N}$ and returns the updated list. For p larger than the list size, we assume an insertion at the end. We assume that all inserted elements are unique, which can be achieved by attaching replica identifiers and sequence numbers.

- $\textsc{Del}(a, p)$: deletes an element at position $p \in \mathbb{N}$ and returns the updated list. For p larger than the list size, we assume an deletion at the end. The parameter $a \in U$ is used to record the deleted element [22], which will be referred to in condition 1(a) of the weak list specification defined later.
- $\textsc{Read}$: returns the contents of the list.

The operations above, as well as a special NOP (i.e., "do nothing"), form $\mathcal{O}$ and all possible list contents form *Val*. $\textsc{Ins}$ and $\textsc{Del}$ are collectively called *list updates*. We denote by $\text{elems}(A) = \{a \mid \text{do}(\textsc{Ins}(a, _), _) \in H\}$ the set of all elements inserted into the list in an abstract execution $A = (H, \text{vis})$.

We adopt the convergence property in [5] which requires that two $\textsc{Read}$ operations that observe the same set of list updates return the same response. Formally, an abstract execution $A = (H, \text{vis})$ belongs to the *convergence property* $\mathcal{A}_{\text{cp}}$ if and only if for any pair of $\textsc{Read}$ events $e_1 = \text{do}(\textsc{Read}, w_1 \triangleq a_1^0 \dots a_1^{m-1})$ and $e_2 = \text{do}(\textsc{Read}, w_2 \triangleq a_2^0 \dots a_2^{n-1})$ ($a_i^j \in \text{elems}(A)$), it holds that $\left(\text{vis}^{-1}_{\textsc{Ins},\textsc{Del}}(e_1) = \text{vis}^{-1}_{\textsc{Ins},\textsc{Del}}(e_2)\right) \implies w_1 = w_2$, where $\text{vis}^{-1}_{\textsc{Ins},\textsc{Del}}(e)$ denotes the set of list updates visible to e.

The weak list specification requires the ordering between elements that are not deleted to be consistent across the system [5].

▶ **Definition 1** (Weak List Specification $\mathcal{A}_{\text{weak}}$ [5])**.** An abstract execution $A = (H, \text{vis})$ belongs to the *weak list specification* $\mathcal{A}_{\text{weak}}$ if and only if there is a relation $\text{lo} \subseteq \text{elems}(A) \times \text{elems}(A)$, called the *list order*, such that:

1. Each event $e = \text{do}(o, w) \in H$ returns a sequence of elements $w = a_0 \dots a_{n-1}$, where $a_i \in \text{elems}(A)$, such that:
   a. w contains exactly the elements visible to e that have been inserted, but not deleted:
   $$\forall a.\, a \in w \iff \left(\text{do}(\textsc{Ins}(a, _), _) \le_{\text{vis}} e\right) \land \neg\left(\text{do}(\textsc{Del}(a, _), _) \le_{\text{vis}} e\right).$$
   b. The list order is consistent with the order of the elements in w:
   $$\forall i, j.\, (i < j) \implies (a_i, a_j) \in \text{lo}.$$
   c. Elements are inserted at the specified position: $op = \textsc{Ins}(a, k) \implies a = a_{\min\{k, n-1\}}$.
2. lo is irreflexive and for all events $e = \text{do}(op, w) \in H$, it is transitive and total on $\{a \mid a \in w\}$.

▶ **Example 2** (Weak List Specification)**.** In the execution depicted in Figure 1 (produced by CJupiter), there exist three states with list contents $w_1 = ba$, $w_2 = ax$, and $w_3 = xb$, respectively. This is allowed by the weak list specification with the list order lo: $b \xrightarrow{\text{lo}} a$ on w_1, $a \xrightarrow{\text{lo}} x$ on w_2, and $x \xrightarrow{\text{lo}} b$ on w_3. However, an execution is not allowed by the weak list specification if it contained two states with, say $w = ab$ and $w' = ba$.

2.4 Operational Transformation (OT)

The OT of transforming $o_1 \in \mathcal{O}$ with $o_2 \in \mathcal{O}$ is expressed by the function $o_1' = OT(o_1, o_2)$. We also write $(o_1', o_2') = OT(o_1, o_2)$ to denote both $o_1' = OT(o_1, o_2)$ and $o_2' = OT(o_2, o_1)$. To ensure the convergence property, OT functions are required to satisfy CP1 (Convergence Property 1) [8]: Given two operations o_1 and o_2, if $(o_1', o_2') = OT(o_1, o_2)$, then $\sigma; o_1; o_2' = \sigma; o_2; o_1'$ should hold, meaning that the same state is obtained by applying o_1 and o_2' in sequence, and applying o_2 and o_1' in sequence, on the same initial state σ. A set of OT functions satisfying CP1 for a replicated list object [8, 9, 22] can be found in Figure A.1 of [25].

3 The CJupiter Protocol

In this section we propose CJupiter (Compact Jupiter) for a replicated list based on the data structure called n-ary ordered state space. Like Jupiter, CJupiter also adopts a client/server architecture. For convenience, we assume that the server does not generate operations [26, 5]. It mainly serializes operations and propagates them from one client to others. We denote by '$\prec_s$' the total order on the set of operations established by the server. Note that '$\prec_s$' is consistent with the causally-before relation '$\xrightarrow{\text{hb}}$'. To facilitate the comparison of Jupiter and CJupiter, we refer to '$\xrightarrow{\text{hb}}$' and '$\prec_s$' together as the *schedule* of operations.

3.1 Data Structure: n-ary Ordered State Space

For a client/server system with n clients, CJupiter maintains $(n+1)$ n-ary ordered state spaces, one per replica (CSS_s for the server and CSS_{c_i} for client c_i). Each CSS is a directed graph whose vertices represent states and edges are labeled with operations; see Appendix B.1 of [25].

An ***operation*** op of type Op is a tuple $op = (o, oid, ctx, sctx)$, where *1)* o is the signature of type $\mathcal{O}$ described in Section 2.3; *2)* oid is a globally unique operation identifier which is a pair (cid, seq) consisting of the client id and a sequence number; *3)* ctx is an *operation context* which is a set of oids, denoting the operations that are causally before op; and *4)* $sctx$ is a set of oids, denoting the operations that, as far as op knows, have been executed before op at the server. At a given replica, $sctx$ is used to determine the total order '$\prec_s$' relation between two operations as in Algorithm B.1 of [25].

The OT function of two operations $op, op' \in Op$, denoted $(op\langle op'\rangle : Op, op'\langle op\rangle : Op) = OT(op, op')$, is defined based on that of $op.o, op'.o \in \mathcal{O}$, denoted $(o, o') = OT(op.o, op'.o)$, such that $op\langle op'\rangle = (o, op.oid, op.ctx \cup \{op'.oid\}, op.sctx)$ and $op'\langle op\rangle = (o', op'.oid, op'.ctx \cup \{op.oid\}, op'.sctx)$.

A ***vertex*** v of type $Vertex$ is a pair $v = (oids, edges)$, where $oids$ is the set of operations (represented by their identifies) that have been executed, and $edges$ is an *ordered* set of edges of type $Edge$ from v to other vertices, labeled with operations. That is, each ***edge*** is a pair $(op : Op, v : Vertex)$. Edges from the same vertex are *totally ordered* by their op components. For each vertex v and each edge $e = (op, u)$ from v to u, it is required that

- the ctx of op associated with e matches the $oids$ of v: $op.ctx = v.oids$;
- the $oids$ of u consists of the $oids$ of v and the oid of op: $u.oids = v.oids \cup \{op.oid\}$.

▶ **Definition 3** (n-ary Ordered State Space)**.** An ***n-ary ordered state space*** is a set of vertices such that

1. Vertices are uniquely identified by their $oids$.
2. For each vertex u with $|u.edges| \geq 2$, let u' be its child vertex along the ***first*** edge $e_{uu'} = (op', u')$ and v another child vertex along $e_{uv} = (op, v)$. There exist (Figure 3)
   - a vertex v' with $v'.oids = u.oids \cup \{op'.oid, op.oid\}$;
   - two edges $e_{u'v'} = (op\langle op'\rangle, v')$ from u' to v' and $e_{vv'} = (op'\langle op\rangle, v')$ from v to v'.

The second condition models OTs in CJupiter described in Section 3.2, and the choice of the "first" edge is justified in Lemmas 5 and 7.

3.2 The CJupiter Protocol

Each replica in CJupiter maintains an n-ary ordered state space S and keeps the most recent vertex cur (initially $(\emptyset, \emptyset)$) of S. Following [26], we describe CJupiter in three parts; see Appendix B.2 of [25] for pseudocode.

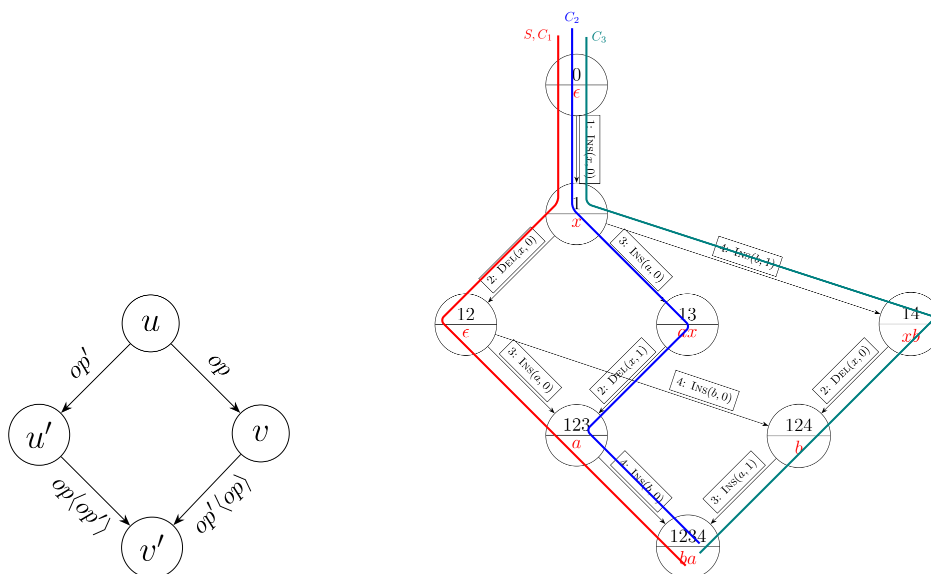


Figure 3 Illustration of an OT of two operations op, op' in both the n-ary ordered state space of CJupiter and the 2D state space of Jupiter: $(op\langle op'\rangle, op'\langle op\rangle) = OT(op, op')$. In the CJupiter and Jupiter protocols (and Examples 4 and 13), op corresponds to the new incoming operation to be transformed.

Figure 4 The same final n-ary ordered state space (thus for CSS_s and each CSS_{c_i}) constructed by CJupiter for each replica under the schedule of Figure 1. Each replica behavior (i.e., the sequence of state transitions) corresponds to a path going through this state space.

Local Processing Part. When a client receives an operation $o \in \mathcal{O}$ from a user, it

1. applies o locally, obtaining a new list $val \in Val$;
2. generates $op \in Op$ by attaching to o a unique operation identifier and the operation context $S.cur.oids$, representing the set of operations that are causally before op;
3. creates a vertex v with $v.oids = S.cur.oids \cup \{op.oid\}$, appends v to S by linking it to $S.cur$ via an edge labeled with op, and updates cur to be v;
4. sends op to the server asynchronously and returns val to the user.

Server Processing Part. To establish the total order '$\prec_s$' on operations, the server maintains the set $soids$ of operations it has executed. When the server receives an operation $op \in Op$ from client c_i, it

1. updates $op.sctx$ to be $soids$ and updates $soids$ to include $op.oid$;
2. transforms op with an operation sequence in S to obtain op' by calling $S.\textsc{xForm}(op)$ (see below), and applies op' (specifically, $op'.o$) locally;
3. sends op (instead of op') to other clients asynchronously.

Remote Processing Part. When a client receives an operation $op \in Op$ from the server, it transforms op with an operation sequence in S to obtain op' by calling $S.\textsc{xForm}(op)$ (see below), and applies op' (specifically, $op'.o$) locally.

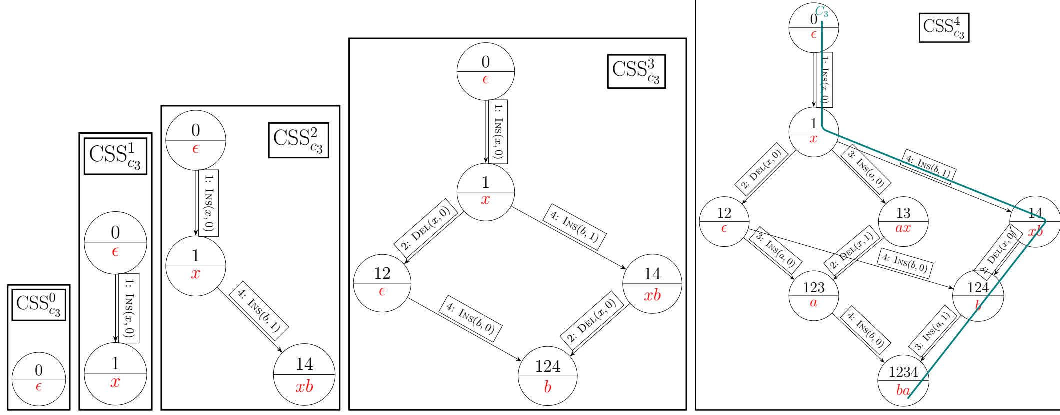


Figure 5 Illustration of client c_3 in CJupiter under the schedule of Figure 1. Its behavior (i.e., the sequence of state transitions) is indicated by the path in CSS$^4_{c_3}$. (Please refer to Figure B.1 of [25] for the illustration of clients c_1 and c_2 and the server s.)

OTs in CJupiter. The procedure $S.\text{xForm}(op : Op)$ transforms op with an operation sequence in an n-ary ordered state space S. Specifically, it

1. locates the vertex u whose $oids$ matches the ctx of op, i.e., $u.oids = op.ctx$ [4], and creates a vertex v with $v.oids = u.oids \cup \{op.oid\}$;
2. iteratively transforms op with an operation sequence consisting of operations along the ***first*** edges from u to the final vertex cur of S (Figure 3):
   a. obtains the vertex u' and the operation op' associated with the first edge of u;
   b. transforms op with op' to obtain $op\langle op'\rangle$ and $op'\langle op\rangle$;
   c. creates a vertex v' with $v'.oids = v.oids \cup \{op'.oid\}$;
   d. links v' to v via an edge labeled with $op'\langle op\rangle$ and v to u via an edge labeled with op;
   e. updates u, v, and op to be u', v', and $op\langle op'\rangle$, respectively;
3. when u is the final vertex cur of S, links v to u via an edge labeled with op, updates cur to be v, and returns the last transformed operation op.

To keep track of the construction of the n-ary ordered state spaces in CJupiter, for each state space, we introduce a superscript k to refer to the one after the k-th step (i.e., after processing k operations), counting from 0. For instance, the state space CSS$_{c_i}$ (resp. CSS$_s$) after the k-th step maintained by client c_i (resp. the server s) is denoted by CSS$^k_{c_i}$ (resp. CSS^k_s). This notational convention also applies to Jupiter (reviewed in Section 4.1).

▶ **Example 4** (Illustration of CJupiter)**.** Figure 5 illustrates client c_3 in CJupiter under the schedule of Figure 1. For convenience, we denote, for instance, a vertex v with $v.oids = \{o_1, o_4\}$ by v_{14} and an operation o_3 with $o_3.ctx = \{o_1, o_2\}$ by $o_3\{o_1, o_2\}$. We have also mixed the notations of operations of types $\mathcal{O}$ and Op when no confusion arises. We map various vertices and operations in this example to the ones (i.e., u, u', v, v', op, op') used in the description of the CJupiter protocol.

After receiving and applying $o_1 = \text{Ins}(x, 0)$ of client c_1 from the server, client c_3 generates $o_4 = \text{Ins}(b, 1)$. It applies o_4 locally, creates a new vertex v_{14}, and appends it to CSS$^1_{c_3}$ via an edge from v_1 labeled with $o_4\{o_1\}$. Then, $o_4\{o_1\}$ is propagated to the server.

[4] The vertex u exists due to the FIFO communication between the clients and the server.

Next, client c_3 receives $o_2 = \text{Del}(x, 0)$ of client c_1 from the server. The operation context of o_2 is $\{o_1\}$, matching the *oids* of v_1 (u). By xForm, $o_2\{o_1\}$ (op) is transformed with $o_4\{o_1\}$ (op'): $OT\Big(o_2\{o_1\} = \text{Del}(x,0), o_4\{o_1\} = \text{Ins}(b,1)\Big) = \Big(o_2\{o_1, o_4\} = \text{Del}(x,0), o_4\{o_1, o_2\} = \text{Ins}(b,0)\Big)$. As a result, v_{124} (v') is created and is linked to v_{12} (v) and v_{14} (u') via the edges labeled with $o_4\{o_1, o_2\}$ and $o_2\{o_1, o_4\}$, respectively. Because o_2 is unaware of o_4 at the server ($o_4.sctx = \emptyset$ now), the edge from v_1 to v_{12} is ordered before (to the left of) that from v_1 to v_{14} in $\text{CSS}^3_{c_3}$.

Finally, client c_3 receives $o_3\{o_1\} = \text{Ins}(a, 0)$ of client c_2 from the server. The operation context of o_3 is $\{o_1\}$, matching the *oids* of v_1 (u). By xForm, $o_3\{o_1\}$ will be transformed with the operation sequence consisting of operations along the *first* edges from v_1 to the final vertex v_{124} of $\text{CSS}^3_{c_3}$, namely $o_2\{o_1\}$ from v_1 and $o_4\{o_1, o_2\}$ from v_{12}. Specifically, $o_3\{o_1\}$ (op) is first transformed with $o_2\{o_1\}$ (op'): $OT\Big(o_3\{o_1\} = \text{Ins}(a,0), o_2\{o_1\} = \text{Del}(x,0)\Big) = \Big(o_3\{o_1, o_2\} = \text{Ins}(a,0), o_2\{o_1, o_3\} = \text{Del}(x,1)\Big)$. Since o_3 is aware of o_2 but unaware of o_4 at the server, the new edge from v_1 labeled with $o_3\{o_1\}$ is placed before that with $o_4\{o_1\}$ but after that with $o_2\{o_1\}$. Then, $o_3\{o_1, o_2\}$ (op) is transformed with $o_4\{o_1, o_2\}$ (op'), yielding v_{1234} and $o_3\{o_1, o_2, o_4\}$. Client c_3 applies $o_3\{o_1, o_2, o_4\}$, obtaining the list content *ba*.

The choice of the "first" edges in OTs is necessary to establish equivalence between CJupiter and Jupiter, particularly *at the server side*. First, the operation sequence along the first edges from a vertex of CSS_s at the server admits a simple characterization.

▶ **Lemma 5** (CJupiter's "First" Rule). *Let $OP = \langle op_1, op_2, \ldots, op_m \rangle$ ($op_i \in Op$) be the operation sequence the server has currently processed in total order '$\prec_s$'. For any vertex v in the current CSS_s, the path along the **first** edges from v to the final vertex of CSS_s consists of the operations of $OP \setminus v$ in total order '$\prec_s$' (may be empty if v is the final vertex of CSS_s), where*

$$OP \setminus v = \Big\{ op \in OP \mid op.oid \in \{op_1.oid, op_2.oid, \cdots, op_m.oid\} \setminus v.oids \Big\}.$$

▶ **Example 6** (CJupiter's "First" Rule). Consider CSS_s at the server shown in Figure 4 under the schedule of Figure 1; see Figure B.1a of [25] for its construction. Suppose that the server has processed all four operations. That is, we take $OP = \langle o_1, o_2, o_3, o_4 \rangle$ in Lemma 5 (we mix operations of types $\mathcal{O}$ and Op). Then, the path along the first edges from vertex v_1 (resp. v_{13}) consists of the operations $OP \setminus v_1 = \{o_2, o_3, o_4\}$ (resp. $OP \setminus v_{13} = \{o_2, o_4\}$) in total order '$\prec_s$'.

Based on Lemma 5, the operation sequence with which an operation transforms *at the server* can be characterized as follows, which is exactly the same with that for Jupiter [26].

▶ **Lemma 7** (CJupiter's OT Sequence). *In* xForm *of* CJupiter*, the operation sequence L (may be empty) with which an operation op transforms **at the server** consists of the operations that are both totally ordered by '$\prec_s$' before and concurrent by '$\|$' with op. Furthermore, the operations in L are totally ordered by '$\prec_s$'.*

▶ **Example 8** (CJupiter's OT Sequence). Consider the behavior of the server summarized in Figure 4 under the schedule of Figure 1. According to Lemma 5, the operation sequence with which $op = o_4$ transforms consists of operations o_2 (i.e., $o_2\{o_1\}$) from vertex v_1 and o_3 (i.e., $o_3\{o_1, o_2\}$) from vertex v_{12} in total order '$\prec_s$', which are both totally ordered by '$\prec_s$' before and concurrent by '$\|$' with o_4.

3.3 CJupiter is Compact

Although $(n+1)$ n-ary ordered state spaces are maintained by CJupiter for a system with n clients, they are all the same. That is, at a high level, CJupiter maintains only a single n-ary ordered state space.

▶ **Proposition 9** $(n+1 \Rightarrow 1)$**.** *In* CJupiter*, the replicas that have processed the same set of operations (in terms of their oids) have the same n-ary ordered state space.*

Informally, this proposition holds because we have kept all "by-product" states/vertices of OTs in the n-ary ordered state spaces, and each client is "synchronized" with the server. Since all replicas will eventually process all operations, the final n-ary ordered state spaces at all replicas are the same. The construction order may differ replica by replica.

▶ **Example 10** (CJupiter is Compact)**.** Figure 4 shows the same final n-ary ordered state space constructed by CJupiter for each replica under the schedule of Figure 1. (Figure B.1 of [25] shows the step-by-step construction for each replica.) Each replica behavior (i.e., the sequence of state transitions) corresponds to a path going through this state space. As illustrated, the server s and client c_1 go along the path $v_0 \xrightarrow{o_1} v_1 \xrightarrow{o_2} v_{12} \xrightarrow{o_3} v_{123} \xrightarrow{o_4} v_{1234}$, client c_2 goes along the path $v_0 \xrightarrow{o_1} v_1 \xrightarrow{o_3} v_{13} \xrightarrow{o_2} v_{123} \xrightarrow{o_4} v_{1234}$, and client c_3 goes along the path $v_0 \xrightarrow{o_1} v_1 \xrightarrow{o_4} v_{14} \xrightarrow{o_2} v_{124} \xrightarrow{o_3} v_{1234}$.

Together with the fact that the OT functions satisfy CP1, Proposition 9 implies that

▶ **Theorem 11** (CJupiter $\models \mathcal{A}_{\mathsf{cp}}$)**.** CJupiter *satisfies the convergence property $\mathcal{A}_{cp}$.*

4 CJupiter is Equivalent to Jupiter

We now prove that CJupiter is equivalent to Jupiter (reviewed in Section 4.1) from perspectives of both the server and clients. Specifically, we prove that the behaviors of the servers are the same (Section 4.2), and that the behaviors of each pair of corresponding clients are the same (Section 4.3). Consequently, we have that

▶ **Theorem 12** (Equivalence)**.** *Under the same schedule, the behaviors (Section 2.1) of corresponding replicas in* CJupiter *and* Jupiter *are the same.*

4.1 Review of Jupiter

We review the Jupiter protocol in [26], a *multi-client* description of Jupiter first proposed in [13] [5]. Consider a client/server system with n clients. Jupiter [26] maintains $2n$ *2D state spaces* (Appendix C.1 of [25]), each consisting of a *local* dimension and a *global* dimension. Specifically, each client c_i maintains a 2D state space, denoted DSS_{c_i}, with the local dimension for operations generated by the client and the global dimension by others. The server maintains n 2D state spaces, one for each client. The state space for client c_i, denoted DSS_{s_i}, consists of the local dimension for operations from client c_i and the global dimension from others.

Jupiter is similar to CJupiter with two major differences: First, in $\textsc{xForm}(op : Op, d \in \{\mathit{LOCAL}, \mathit{GLOBAL}\})$ of Jupiter, the operation sequence with which op transforms is determined by the parameter d, indicating the local/global dimension described above (instead of following

[5] The Jupiter protocol in [13] uses 1D buffers, but does not explicitly describe the multi-client scenario.

the *first* edges as in CJupiter). Second, in Jupiter, the server propagates the *transformed* operation (instead of the original one it receives) to other clients. As with CJupiter, we describe Jupiter in three parts. We omit the details that are in common with and have been explained in CJupiter; see Appendix C.2 of [25] for pseudocode.

Local Processing Part. When client c_i receives an operation $o \in \mathcal{O}$ from a user, it applies o locally, generates $op \in Op$ for o, saves op along the local dimension at the end of its 2D state space DSS_{c_i}, and sends op to the server asynchronously.

Server Processing Part. When the server receives an operation $op \in Op$ from client c_i, it first transforms op with an operation sequence along the global dimension in DSS_{s_i} to obtain op' by calling $\text{xForm}(op, \mathit{GLOBAL})$ (see below), and applies op' locally. Then, for each $j \neq i$, it saves op' at the end of DSS_{s_j} along the global dimension. Finally, op' (instead of op) is sent to other clients asynchronously.

Remote Processing Part. When client c_i receives an operation $op \in Op$ from the server, it transforms op with an operation sequence along the local dimension in its 2D state space DSS_{c_i} to obtain op' by calling $\text{xForm}(op, \mathit{LOCAL})$ (see below), and applies op' locally.

OTs in Jupiter. In the procedure $\text{xForm}(op : Op, d : LG = \{\mathit{LOCAL}, \mathit{GLOBAL}\})$ of Jupiter, the operation sequence with which op transforms is determined by an extra parameter d. Specifically, it first locates the vertex u whose *oids* matches the operation context $op.ctx$ of op, and then iteratively transforms op with an operation sequence along the d dimension from u to the final vertex of this 2D state space.

▶ **Example 13** (Illustration of Jupiter). Figure 6 illustrates client c_3, as well as the server s, in Jupiter under the schedule of Figure 1. The first three state transitions made by client c_3 in Jupiter due to the operation sequence consisting of o_1 from client c_1, o_4 generated by itself, and o_2 from client c_1 are the same with those in CJupiter; see $\text{CSS}^1_{c_3}$, $\text{CSS}^2_{c_3}$, and $\text{CSS}^3_{c_3}$ of Figure 5 and $\text{DSS}^1_{c_3}$, $\text{DSS}^2_{c_3}$, and $\text{DSS}^3_{c_3}$ of Figure 6.

We now elaborate on the fourth state transition of client c_3 in Jupiter. First, client c_2 propagates its operation $o_3\{o_1\} = \text{Ins}(a, 0)$ to the server s. At the server, $o_3\{o_1\}$ is transformed with $o_2\{o_1\} = \text{Del}(x, 0)$ in $\text{DSS}^3_{s_2}$, obtaining $o_3\{o_1, o_2\} = \text{Ins}(a, 0)$. In addition to being stored in $\text{DSS}^3_{s_1}$ and $\text{DSS}^3_{s_3}$, the transformed operation $o_3\{o_1, o_2\}$ is then redirected by the server to clients c_1 and c_3. At client c_3, the operation context of $o_3\{o_1, o_2\}$ (i.e., $\{o_1, o_2\}$) matches the *oids* of v_{12} (u) in $\text{DSS}^4_{c_3}$. By xForm, $o_3\{o_1, o_2\}$ (op) is transformed with $o_4\{o_1, o_2\}$ (op'), yielding v_{1234} and $o_3\{o_1, o_2, o_4\}$. Finally, client c_3 applies $o_3\{o_1, o_2, o_4\}$, obtaining the list content *ba*.

We highlight three differences between CJupiter and Jupiter, by comparing the behaviors of client c_3 in this example and Example 4. First, the fourth operation the server s redirects to client c_3 is the transformed operation $o_3\{o_1, o_2\} = \text{Ins}(a, 0)$, instead of the original one $o_3\{o_1\} = \text{Ins}(a, 0)$ [6] generated by client c_2. Second, each vertex in the n-ary ordered state space of CJupiter (such as $\text{CSS}^4_{c_3}$ of Figure 5) is not restricted to have only two child vertices, while Jupiter does. Third, because the transformed operations are propagated by the server, Jupiter is slightly optimized in implementation *at clients* by eliminating redundant OTs. For example, in $\text{CSS}^4_{c_3}$ of Figure 5, the original operation $o_3\{o_1\}$ of client c_2 redirected by the

[6] Although they happen to have the same signature $\text{Ins}(a, 0)$, they have different operation contexts.

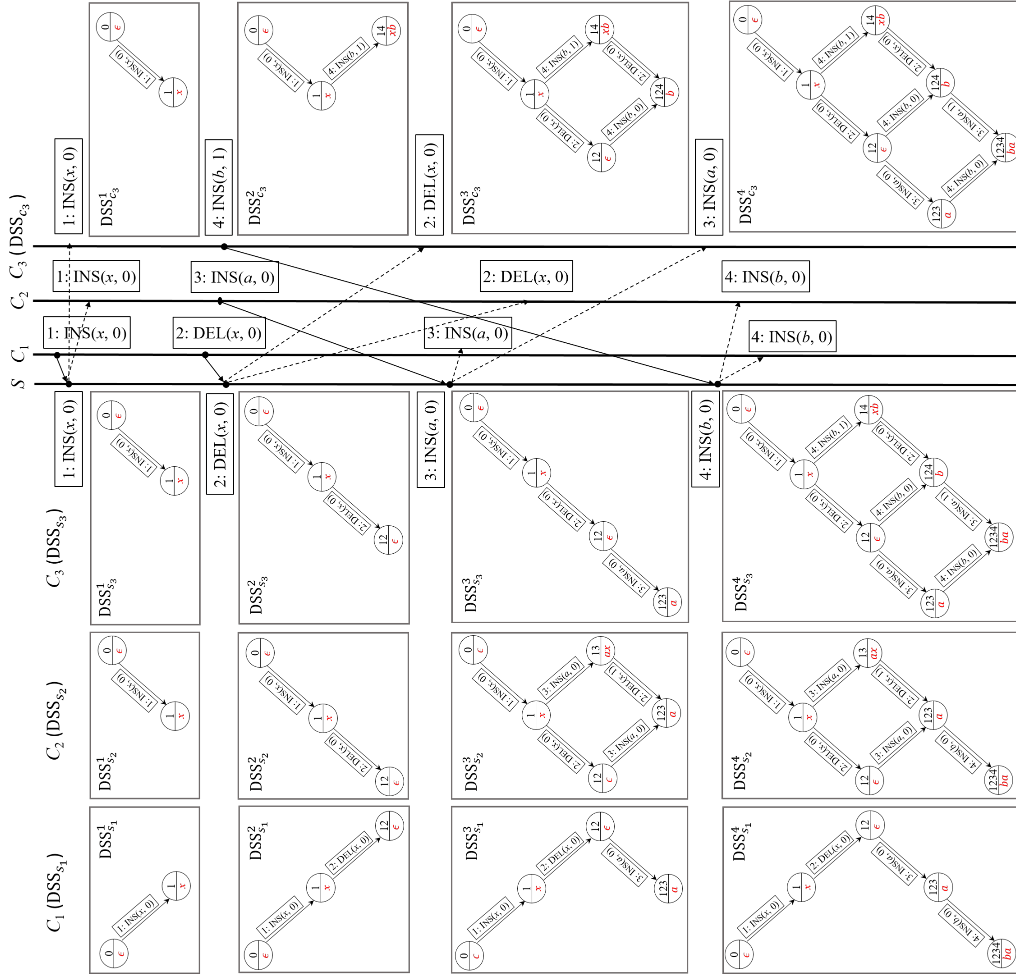


Figure 6 (Rotated) illustration of client c_3, as well as the server s, in Jupiter [26] under the schedule of Figure 1. (Please refer to Figure C.1 of [25] for details of clients c_1 and c_2.)

server should be first transformed with $o_2\{o_1\}$ to obtain $o_3\{o_1, o_2\}$. In Jupiter, however, such a transformation which has been done at the server (i.e., in $\text{DSS}^3_{s_2}$) is not necessary at client c_3 (i.e., in $\text{DSS}^4_{c_3}$).

4.2 The Servers Established Equivalent

As shown in [26] (see the "Jupiter" section and Definition 8 of [26]), the operation sequence with which an incoming operation transforms *at the server* in xForm of Jupiter can be characterized exactly as in xForm of CJupiter (Lemma 7). By mathematical induction on the operation sequence the server processes, we can prove that the state spaces of Jupiter and CJupiter at the server are essentially the same. Formally, the n-ary ordered state space CSS_s of CJupiter equals the union [7] of all 2D state spaces DSS_{s_i} maintained at the server for each client c_i in Jupiter. For example, CSS_s of Figure 4 is the union of the three DSS_{s_i}'s of Figure 6. More specifically, we have

[7] The union is taken on state spaces which are (directed) graphs as sets of vertices and edges. The order of edges of n-ary ordered state spaces should be respected when DSS_{s_i}'s are unioned to obtain CSS_s.

▶ **Proposition 14** ($n \leftrightarrow 1$). *Suppose that under the same schedule, the server has processed a sequence of m operations, denoted $O = \langle op_1, op_2, \ldots, op_m \rangle$ ($op_i \in Op$), in total order '$\prec_s$'. We have that*

$$CSS_s^k = \bigcup_{i=1}^{i=k} DSS_{s_{c(op_i)}}^i = \bigcup_{c_i \in c(O)} \bigcup_{j=1}^{j=k} DSS_{s_{c_i}}^j, \quad 1 \leq k \leq m, \tag{$*$}$$

where $c(op_i)$ denotes the client that generates the operation op_i (more specifically, $op_i.o$) and $c(O) = \{c(op_1), c(op_2), \ldots, c(op_m)\}$.

The equivalence of servers are thus established.

▶ **Theorem 15** (Equivalence of Servers). *Under the same schedule, the behaviors (i.e., the sequence of (list) state transitions, defined in Section 2.1) of the servers in* CJupiter *and* Jupiter *are the same.*

4.3 The Clients Established Equivalent

As discussed in Example 13, Jupiter is slightly optimized in implementation *at clients* by eliminating redundant OTs. Formally, by mathematical induction on the operation sequence client c_i processes, we can prove that $\mathrm{DSS}_{c_i}^k$ of Jupiter is a part (i.e., subgraph) of $\mathrm{CSS}_{c_i}^k$ of CJupiter. The equivalence of clients follows since the final transformed operations (for an original one) executed at c_i in Jupiter and CJupiter are the same, regardless of the optimization adopted by Jupiter at clients.

▶ **Proposition 16** ($1 \leftrightarrow 1$). *Under the same schedule, we have that*

$$DSS_{c_i}^k \subseteq CSS_{c_i}^k, \quad 1 \leq i \leq n,\ k \geq 1. \tag{$\star$}$$

▶ **Theorem 17** (Equivalence of Clients). *Under the same schedule, the behaviors (Section 2.1) of each pair of corresponding clients in* CJupiter *and* Jupiter *are the same.*

5 CJupiter Satisfies the Weak List Specification

The following theorem, together with Theorem 12, solves the conjecture of Attiya et al. [5].

▶ **Theorem 18** (CJupiter $\models \mathcal{A}_{\text{weak}}$). CJupiter *satisfies the weak list specification $\mathcal{A}_{weak}$.*

Proof. For each execution α of CJupiter, we construct an abstract execution $A = (H, \mathrm{vis})$ with $\mathrm{vis} = \xrightarrow{\mathrm{hb}_\alpha}$ (Section 2.1). We then prove the conditions of $\mathcal{A}_{\text{weak}}$ (Definition 1) in the order 1(c), 1(a), 1(b), and 2.

Condition 1(c) follows from the local processing of CJupiter. Condition 1(a) holds due to the FIFO communication and the property of OTs that when transformed in CJupiter, the type and effect of an $\textsc{Ins}(a, p)$ (resp. a $\textsc{Del}(a, p)$) remains unchanged (with a trivial exception of being transformed to be NOP), namely to insert (resp. delete) the element a (possibly at a different position than p).

To show that $A = (H, \mathrm{vis})$ belongs to $\mathcal{A}_{\text{weak}}$, we define the list order relation lo in Definition 19 below, and then prove that lo satisfies conditions 1(b) and 2 of Definition 1. ◀

▶ **Definition 19** (List Order 'lo'). Let α be an execution. For $a, b \in \mathrm{elems}(A)$, $a \xrightarrow{\mathrm{lo}} b$ if and only if there exists an event $e \in \alpha$ with returned list w such that a precedes b in w.

By definition, *1)* lo is *transitive* and *total* on $\{a \mid a \in w\}$ for all events $e = \text{do}(o, w) \in H$; *and 2)* lo satisfies 1(b) of Definition 1. The *irreflexivity* of lo can be rephrased in terms of the pairwise state compatibility property.

▶ **Definition 20** (State Compatibility). Two list states w_1 and w_2 are *compatible*, if and only if for any two common elements a and b of w_1 and w_2, their relative orderings are the same in w_1 and w_2.

▶ **Lemma 21** (Irreflexivity). *Let α be an execution and $A = (H, vis)$ the abstract execution constructed from α as described in the proof of Theorem 18. The list order* lo *based on α is irreflexive if and only if the list states (i.e., returned lists) in A are pairwise compatible.*

The proof relies on the following lemma about paths in n-ary ordered state spaces.

▶ **Lemma 22** (Simple Path). *Let $P_{v_1 \rightsquigarrow v_2}$ be a path from vertex v_1 to vertex v_2 in an n-ary ordered state space. Then, there are no duplicate operations (in terms of their oids) along the path $P_{v_1 \rightsquigarrow v_2}$. We call such a path a simple path.*

Therefore, it remains to prove that all list states in an execution of CJupiter are pairwise compatible, which concludes the proof of Theorem 18. By Proposition 9, we can focus on the state space CSS_s at the server. We first prove several properties about vertex pairs and paths of CSS_s, which serve as building blocks for the proof of the main result (Theorem 26).

By mathematical induction on the operation sequence processed in the total order '$\prec_s$' at the server and by contradiction (in the inductive step), we can show that

▶ **Lemma 23** (LCA). *In* CJupiter*, each pair of vertices in the n-ary ordered state space CSS_s (as a rooted directed acyclic graph) has a unique LCA (Lowest Common Ancestor).* [8]

In the following, we are concerned with the paths to a pair of vertices from their LCA.

▶ **Lemma 24** (Disjoint Paths). *Let v_0 be the unique LCA of a pair of vertices v_1 and v_2 in the n-ary ordered state space CSS_s, denoted $v_0 = LCA(v_1, v_2)$. Then, the set of operations $O_{v_0 \rightsquigarrow v_1}$ along a simple path $P_{v_0 \rightsquigarrow v_1}$ is disjoint in terms of the operation oids from the set of operations $O_{v_0 \rightsquigarrow v_2}$ along a simple path $P_{v_0 \rightsquigarrow v_2}$.*

The next lemma gives a sufficient condition for two states (vertices) being compatible in terms of disjoint simple paths to them from a common vertex.

▶ **Lemma 25** (Compatible Paths). *Let $P_{v_0 \rightsquigarrow v_1}$ and $P_{v_0 \rightsquigarrow v_2}$ be two paths from vertex v_0 to vertices v_1 and v_2, respectively in the n-ary ordered state space CSS_s. If they are disjoint simple paths, then the list states of v_1 and v_2 are compatible.*

The desired pairwise state compatibility property follows, when we take the common vertex v_0 in Lemma 25 as the LCA of the two vertices v_1 and v_2 under consideration.

▶ **Theorem 26** (Pairwise State Compatibility). *Every pair of list states in the state space CSS_s are compatible.*

Proof. Consider vertices v_1 and v_2 in CSS_s. *1)* By Lemma 23, they have a unique LCA, denoted v_0; *2)* By Lemma 22, $P_{v_0 \rightsquigarrow v_1}$ and $P_{v_0 \rightsquigarrow v_2}$ are simple paths; *3)* By Lemma 24, $P_{v_0 \rightsquigarrow v_1}$ and $P_{v_0 \rightsquigarrow v_2}$ are disjoint; and *4)* By Lemma 25, the list states of v_1 and v_2 are compatible. ◀

[8] The LCAs of two vertices v_1 and v_2 in a rooted directed acyclic graph is a set of vertices V such that *1)* Each vertice in V has both v_1 and v_2 as descendants; *2)* In V, no vertice is an ancestor of another. The uniqueness further requires $|V| = 1$.

6 Related Work

Convergence is the main property for implementing a highly-available replicated list object [8, 26]. Since 1989 [8], a number of OT [8]-based protocols have been proposed. These protocols can be classified according to whether they rely on a total order on operations [26]. Various protocols like Jupiter [13, 26] establish a total order via a central server, a sequencer, or a distributed timestamping scheme [1, 24, 18, 12, 23]. By contrast, protocols like adOPTed [15] rely only on a partial (causal) order on operations [8, 14, 21, 20, 19].

In 2016, Attiya et al. [5] propose the strong/weak list specification of a replicated list object. They prove that the existing CRDT (Conflict-free Replicated Data Types) [17]-based RGA protocol [16] satisfies the strong list specification, and *conjecture* that the well-known OT-based Jupiter protocol [13, 26] satisfies the weak list specification.

The OT-based protocols typically use data structures like 1D buffer [18], 2D state space [13, 26], or N-dimensional interaction model [15] to keep track of OTs or choose correct OTs to perform. As a generalization of 2D state space, our n-ary ordered state space is similar to the N-dimensional interaction model. However, they are proposed for different system models. In an n-ary ordered state space, edges from the same vertex are *ordered*, utilizing the existence of a total order on operations. By contrast, the N-dimensional interaction model relies only on a partial order on operations. Consequently, the simple characterization of OTs in xForm of CJupiter does not apply in the N-dimensional interaction model.

7 Conclusion and Future Work

We prove that the Jupiter protocol [13, 26] satisfies the weak list specification [5], thus solving the conjecture recently proposed by Attiya et al. [5]. To this end, we have designed CJupiter based on a novel data structure called n-ary ordered state space. In the future, we will explore how to algebraically manipulate and reason about n-ary ordered state spaces. We also plan to generalize this data structure to the scenarios without a central server and study whether the distributed adOPTed protocol [15] satisfies the strong/weak list specification.

References

1. Apache Wave. `https://incubator.apache.org/wave/`.
2. Google Docs. `https://docs.google.com`.
3. What's different about the new Google Docs: Making collaboration fast. `https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs.html`.
4. *Apache Wave (incubating) Protocol Documentation (Release 0.4)*, August 22, 2015.
5. Hagit Attiya, Sebastian Burckhardt, Alexey Gotsman, Adam Morrison, Hongseok Yang, and Marek Zawirski. Specification and complexity of collaborative text editing. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC '16, pages 259–268. ACM, 2016.
6. Hagit Attiya, Faith Ellen, and Adam Morrison. Limitations of Highly-Available Eventually-Consistent Data Stores. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 385–394. ACM, 2015.
7. Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated Data Types: Specification, Verification, Optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 271–284. ACM, 2014.
8. C. A. Ellis and S. J. Gibbs. Concurrency Control in Groupware Systems. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, SIGMOD '89, pages 399–407. ACM, 1989.

9 Abdessamad Imine, Michaël Rusinowitch, Gérald Oster, and Pascal Molli. Formal Design and Verification of Operational Transformation Algorithms for Copies Convergence. *Theor. Comput. Sci.*, 351(2):167–183, February 2006.

10 Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, July 1978.

11 Bo Leuf and Ward Cunningham. *The Wiki Way: Quick Collaboration on the Web*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

12 Rui Li, Du Li, and Chengzheng Sun. A Time Interval Based Consistency Control Algorithm for Interactive Groupware Applications. In *Proceedings of the 10th International Conference on Parallel and Distributed Systems*, ICPADS '04, pages 429–438, 2004.

13 David A. Nichols, Pavel Curtis, Michael Dixon, and John Lamping. High-latency, Low-bandwidth Windowing in the Jupiter Collaboration System. In *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*, UIST '95, pages 111–120. ACM, 1995.

14 Atul Prakash and Michael J. Knister. A Framework for Undoing Actions in Collaborative Systems. *ACM Trans. Comput.-Hum. Interact.*, 1(4):295–330, December 1994.

15 Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuser. An Integrating, Transformation-oriented Approach to Concurrency Control and Undo in Group Editors. In *Proceedings of the 1996 ACM Conference on Computer Supported Cooperative Work*, CSCW '96, pages 288–297. ACM, 1996.

16 Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. Replicated Abstract Data Types: Building Blocks for Collaborative Applications. *J. Parallel Distrib. Comput.*, 71(3):354–368, March 2011.

17 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free Replicated Data Types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400. Springer-Verlag, 2011.

18 Haifeng Shen and Chengzheng Sun. Flexible Notification for Collaborative Systems. In *Proceedings of the 2002 ACM Conference on Computer Supported Cooperative Work*, CSCW '02, pages 77–86. ACM, 2002.

19 Chengzheng Sun. Undo As Concurrent Inverse in Group Editors. *ACM Trans. Comput.-Hum. Interact.*, 9(4):309–361, December 2002.

20 Chengzheng Sun and Clarence Ellis. Operational Transformation in Real-time Group Editors: Issues, Algorithms, and Achievements. In *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work*, CSCW '98, pages 59–68. ACM, 1998.

21 Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving Convergence, Causality Preservation, and Intention Preservation in Real-time Cooperative Editing Systems. *ACM Trans. Comput.-Hum. Interact.*, 5(1):63–108, March 1998.

22 Chengzheng Sun, Yi Xu, and Agustina Agustina. Exhaustive Search of Puzzles in Operational Transformation. In *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work*, CSCW '14, pages 519–529. ACM, 2014.

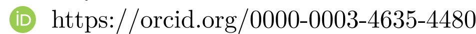
23 David Sun and Chengzheng Sun. Context-Based Operational Transformation in Distributed Collaborative Editing Systems. *IEEE Trans. Parallel Distrib. Syst.*, 20(10):1454–1470, October 2009.

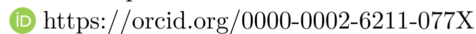
24 Nicolas Vidot, Michelle Cart, Jean Ferrié, and Maher Suleiman. Copies Convergence in a Distributed Real-time Collaborative Environment. In *Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work*, CSCW '00, pages 171–180. ACM, 2000.

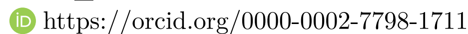
25 Hengfeng Wei, Yu Huang, and Jian Lu. Specification and Implementation of Replicated List: The Jupiter Protocol Revisited. *CoRR*, abs/1708.04754, 2017.

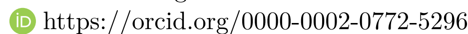
26 Yi Xu, Chengzheng Sun, and Mo Li. Achieving Convergence in Operational Transformation: Conditions, Mechanisms and Systems. In *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work*, CSCW '14, pages 505–518. ACM, 2014.

Local Fast Segment Rerouting on Hypercubes

Klaus-Tycho Foerster
University of Vienna, Vienna, Austria
klaus-tycho.foerster@univie.ac.at

https://orcid.org/0000-0003-4635-4480

Mahmoud Parham[1]
University of Vienna, Vienna, Austria
mahmoud.parham@univie.ac.at

https://orcid.org/0000-0002-6211-077X

Stefan Schmid
University of Vienna, Vienna, Austria
stefan_schmid@univie.ac.at

https://orcid.org/0000-0002-7798-1711

Tao Wen
University of Electronic Science and Technology of China, Chengdu, China
winterwentao@gmail.com

https://orcid.org/0000-0002-0772-5296

Abstract

Fast rerouting is an essential mechanism in any dependable communication network, allowing to quickly, i.e., *locally*, recover from network failures, without invoking the control plane. However, while locality ensures a fast reaction, the absence of global information also renders the design of highly resilient fast rerouting algorithms more challenging. In this paper, we study algorithms for fast rerouting in emerging Segment Routing (SR) networks, where intermediate destinations can be added to packets by nodes along the path. Our main contribution is a maximally resilient polynomial-time fast rerouting algorithm for SR networks based on a hypercube topology. Our algorithm is attractive as it preserves the original paths (and hence waypoints traversed along the way), and does not require packets to carry failure information. We complement our results with an integer linear program formulation for general graphs and exploratory simulation results.

2012 ACM Subject Classification Networks → Routing protocols, Networks → Network reliability, Theory of computation → Design and analysis of algorithms

Keywords and phrases segment routing, local fast failover, link failures

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.13

Acknowledgements We would like to thank David Lebrun for helpful discussions in the early stage of this paper.

1 Introduction

1.1 Motivation and Challenges

The need for a more reliable network performance and quickly growing traffic volumes led, starting from the late 1990s [19], to the development of more advanced approaches to control the routes along which traffic is delivered. Multipath-Label Switching (MPLS) was one of

[1] Contact and main author.

© Klaus-Tycho Foerster, Mahmoud Parham, Stefan Schmid, and Tao Wen;
licensed under Creative Commons License CC-BY
22nd International Conference on Principles of Distributed Systems (OPODIS 2018).
Editors: Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira; Article No. 13; pp. 13:1–13:17

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

the first and most widely deployed alternatives to traditional weight and destination based routing (such as OSPF), enabling a per-flow traffic engineering. Recently, *Segment Routing (SR)* [38, 20] has emerged as a scalable alternative to MPLS networks: SR networks do not require any resource reservations nor states on all the routers part of the route (the *virtual circuit*). SR networks are also attractive for their simple deployment; in contrast to, e.g., Software-Defined Network (SDN) and OpenFlow-based solutions, they rely on existing protocols such as IPv6 [62].

We in this paper investigate how to enhance SR networks with *(local) fast rerouting* algorithms, to react to failures *without the need to invoke the control plane*. The re-computation (and distribution) of routes after failures via the control plane is notoriously slow [26] and known to harm performance [44]. Also link-reversal algorithms [27] tolerating multiple failures have a quadratic convergence time [7], besides requiring dynamic routing tables. This is problematic as certain applications, e.g., in datacenters, are known to require a latency of less than 100 ms [67]; voice traffic [33] and interactive services [35] already degrade after 60 ms of delay. Not surprisingly, reliability is also one of the foremost challenges for network carriers nowadays [66], and in the context of power systems (e.g., smart grids), an almost entirely lossless network is expected [58]. Accordingly, most modern communication networks (including IP, MPLS, OpenFlow networks) feature fast rerouting primitives to support networks to recover quickly from failures.

Designing a fast rerouting algorithm however is non-trivial, as reactions need to be *(statically) pre-defined* and can only depend on the *local* failures, but not on “future” failures, *downstream*. As link failures, also multiple ones, are common in networks [46], e.g., due to shared link risk groups or virtualization, it is crucial to pre-define the conditional local failover rules such that connectivity is preserved (i.e., forwarding loops and blackholes avoided) under *any* possible additional failures. In fact, in many networks, including SR networks, algorithms cannot even depend on already encountered failures *upstream*, as it requires mechanisms to carry and process such information in the packet header; such “failure-carrying packets” [22, 37] require additional and complex forwarding rules. Further challenges are introduced by policy-related constraints on the paths along which packets are rerouted in case of failures. In particular, failover paths may not be allowed to “skip” nodes, but rather should reroute around failed links individually: communication networks include an increasing number of middleboxes and network functions, so-called *waypoints* [2], which must be traversed for security and performance reasons. Without precautions, in case of a link failure, the backup path could omit these waypoints.

Ideally, a local fast rerouting algorithm preserves connectivity “whenever this is still possible”, i.e., as long as the underlying network is still *physically* connected. In other words, in a k-(link-)connected network, we would like the rerouting algorithm to tolerate $k-1$ link failures. We will refer to this strong notion of robustness as *maximal robustness* in the following.

1.2 Example

Figure 1 illustrates an example for the problem considered in this paper: how to efficiently circumvent multiple link failures using SR local fast failover mechanisms, such that the original route will be preserved as part of the packets’ new route, hence also ensuring waypoint traversal. In this example, while the backup path in dotted green reaches the destination, the middlebox w is not visited. Ideally, we want to circumvent the failed link and then continue on the original route (as depicted in the red dashed walk).

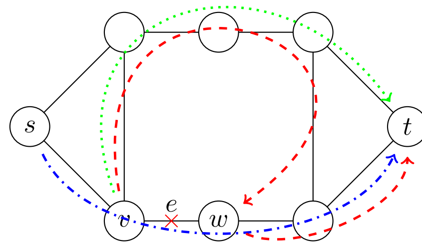


Figure 1 Illustration of two different fast failover mechanisms, upon hitting a link failure. The default path is depicted in dash-dotted blue. In the green dotted path, the destination t is reached, but the waypoint w is not traversed. The red dashed walk circumvents the link failure, traverses w, and then reaches t.

In case of only a single link $e = (v, w)$ failing, one can exploit that both nodes v, w have a globally correct view of the network link states. For example, if a packet hits the failed link e at v, the node v can provide an alternative path to w, after which the packet resumes its original path, as shown in dashed red in Figure 1. To this end, each node only needs to be provisioned with one alternative path for each of its incident links.

In Segment Routing, similar to MPLS, each packet contains a label stack, consisting of nodes or links. However, these labels just represent the next waypoint to be reached, the route ("segment") which depends on the underlying routing functions (e.g., shortest path). Once the top item is reached, the corresponding label is popped and the next item on the stack is parsed. As such, the next label does not need to be in the vicinity of the current node, it can be anywhere in the network. For the case of a single link $e = (v, w)$ failure, it has been shown that pushing two items on the label stack always suffices [25], if the network is still connected and a shortest alternative path is chosen.

While SR enables waypoint traversal even after a single link failure [25], dealing with multiple link failures in SR is still not well understood. As observed in [22], the option of choosing the shortest alternative path already fails under two link failures, see Figure 2; when e_1 fails (and the dash-dotted blue path as well), the packet will be sent along e.g. e_2, but upon the failure of e_2, the packet is sent along e_1—a forwarding loop (shown in dotted red). In this example, we can easily fix the reachability issues: a failure of e_2 causes rerouting along e_3 (in dashed green, not along e_1), and failure of e_3 causes rerouting along e_1. In other words, e_1 depends on e_2, which depends on e_3, which in turn depends on e_1. As this circular dependency chain has a length of three, two failures of $\{e_1, e_2, e_3\}$ cannot induce a forwarding loop when routing to w. We will later formalize and extend these ideas, generating dependency chains of length $\geq k$ for k-dimensional hypercubes.

1.3 Contributions

We initiate the study of fast reroute algorithms for emerging Segment Routing networks which are 1) resilient to a maximum number of failures (i.e., are *maximally robust*), 2) respect the path traversal of the original route, and 3) are compatible to current technologies in that they 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.

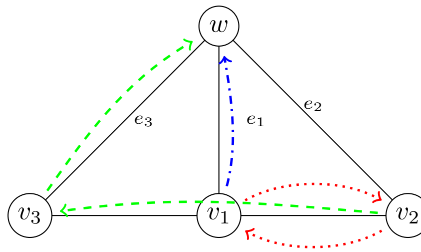


Figure 2 Example illustrating how local fast failover methods for a single link failure can loop under two link failures, as shown in [22]. When the dash-dotted blue default route between v_1 and w fails, v_2 can be pushed as a segment, to in turn reroute along e_2. However, when v_2 uses e_1 via v_1 as a failover for e_2, then failing both e_1 and e_2 leads to a permanent forwarding loop, as depicted in dotted red. In order to route successfully under both e_1, e_2 failing, v_2 has to push segments v_1, v_3 to route along e_3, as depicted in dashed green.

Our main result is an efficient algorithm which provably provides all these properties on hypercube networks, as they are commonly used in datacenters (see e.g., [48]). Furthermore, we formulate the underlying optimization problem as an integer linear program for general graphs, and provide first exploratory insights on the practical performance of segment routing under multiple link failures.

1.4 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 maximally robust SR failover scheme for k-dimensional hypercubes. We cover related work in Section 4 and conclude our study in Section 5, where we also provide further insights which we believe to be useful for future work, in the form of an integer linear program formulation for general graphs and a brief investigation regarding testbed experiments.

2 Model

In this section, we start by providing model and notation preliminaries. We will consider undirected graphs $G = (V, E)$, where the links may be indexed according to some (possibly arbitrary) ordering, with $\ell_i \in E$ denoting the ith link. All routing rules have to be pre-computed and may not be changed during the runtime (e.g., after failures). We will only allow routing rules that match on 1) the packet's next destination (i.e., the top of the label stack)[2], and the 2) incident link failures.[3] When a packet hits a failed link $\ell = (u, v)$ at some node u, the current node u may push a set of pre-computed labels on top of the current label stack, in order to create a so-called backup path to v (which can also be traversed in reverse from v to u).

▶ **Definition 1.** A *backup path* for a link ℓ is a simple path (not containing ℓ) that connects the endpoint of the link ℓ. Let $\mathcal{P}$ be the set of all backup paths in a graph. An injective function $BP : E \to \mathcal{P}$ that maps one backup path to each link is a *backup path scheme*.

[2] In practice, one could also imagine matching on other header fields, such as the packet's source, and also the incoming port. However, our algorithms do not require these additional inputs.
[3] In other words, only the endpoints u, v of the failed link $(u, v) = \ell \in E$ are aware of the failure.

When the packet reaches the current top label, the respective label is popped and the underlying label is set as top label. As such, via backup paths, the incoming packets that normally travel through ℓ are rerouted around the link to the respective endpoint, circumventing the failure. Hence, our model preserves the intermediate visits (i.e., all possible waypoints) and their order in a subset of the traversed route, possibly introducing repeated visits [1]. In the following, we will investigate backup path schemes that guarantee packet delivery even under multiple failures. To this end, we need to ensure that the backup paths do not contain infinite forwarding loops, for their specified maximum number of failures. More formally:

▶ **Definition 2.** A backup path scheme $BP(.)$ is called *f-resilient* if and only if there does not exist a subset of links $L \subseteq E, |L| \leq f$ such that for some ordering $\sigma : \{0, \ldots, |L|-1\} \to \{0, \ldots, |E|-1\}$, $\forall j < |L| : \ell_{\sigma(j+1 \pmod{|L|})} \in BP(\ell_{\sigma(j)})$. We refer to the inclusion relation ($\in$) as *dependency* from $\ell_{\sigma(j)}$ to $\ell_{\sigma(j+1 \pmod{|L|})}$. Equivalently, $BP(.)$ is f-resilient if and only if any *cycle of dependencies* is longer than f.

In the next section, we will show how to efficiently generate a $(k-1)$-resilient backup path scheme for k-dimensional hypercubes. As k-dimensional hypercubes are k-link-connected, our scheme has ideal robustness.

3 Efficient Resilient Segment Routing on k-Dimensional Hypercubes

This section presents a fast and maximally robust rerouting algorithm on hypercubes, one of the most important and well-studied network topologies [53, 64]. The regular structure of hypercubes makes them an ideal fit for e.g., parallel interconnection architectures [52] or datacenter [30].

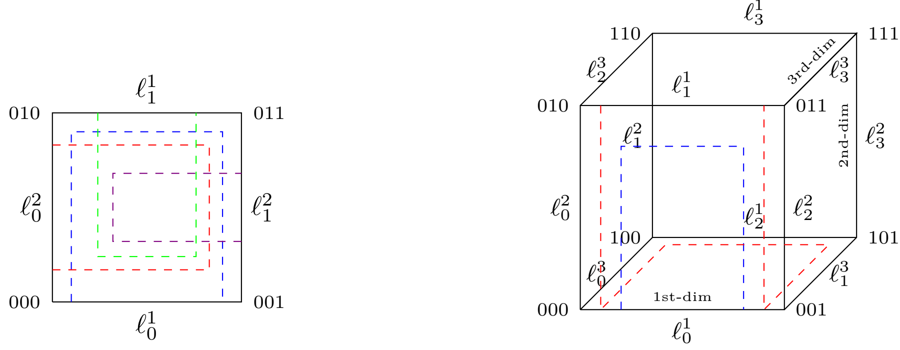
Our study on k-dimensional hypercubes is structured as follows: We first provide an intuition and overview of the $(k-1)$-resilient scheme in Section 3.1, providing a formal definition of all backup paths in Term 1. Next, in Section 3.2, we introduce some useful technical preliminaries for the correctness proof of our scheme, which is presented in Section 3.3.

3.1 Overview of the Fast Local Failover Scheme

We label the nodes in a k-dimensional hypercube (k-*cube*) with tuples $(b_k, b_{k-1}, \ldots, b_1)$, $\forall i \in [k] : b_i \in \{0, 1\}$, such that the *origin* node has the label $\{0\}^k$. A hypercube link is denoted by an ordered pair of binary node labels (a, b) s.t. $a, b \in \{0, 1\}^k, a < b$, where the two labels differ in one bit. Additionally, a link is said to be in dimension $d, d \in [k]$, if and only if a and b differ only at their dth bit. We refer to them as d-*dim* links. For convenience, we treat a hypercube as a set of links grouped by their dimension, within each dimension sorted according to the following bitwise comparison. For $x \in \{0, 1\}^k$, let $x^{>>s} := x >> s$, where $>>$ is right circular shift. Let ℓ_i^d denote the ith link in dimension d, see Figure 3a. For $\ell_p^d = (a, b)$ and $\ell_q^d = (c, d)$, we have $p < q$ if and only if $a^{>>d} < c^{>>d}$. Lastly, we denote a k-cube by $C_k := \cup_{d,i} \ell_i^d, d \in [k], 0 \leq i < 2^{k-1}$.

The idea is to allocate backup paths in k iterations, one for each subset of links in the same dimension, such that the induced dependencies over same-dimension links form cycles of length at least k. However, since there are additional dependency cycles induced by links in different dimensions, we devise a scheme that does not induce any dependency cycle shorter than k (hence $(k-1)$-resiliency follows).

Due to gray coding, starting from any link $\ell_i^d = (a, b)$, by traversing the (unique) pair of incident d'-dim links, we reach the link $N_{d'}(\ell_i^d) = (a', b')$ such that $a' = (2^{d'-1})_2 \oplus a$



(a) 1-resilient scheme for the 2-cube, backup paths shown with dashed lines.

(b) Link and node labels for the 3-cube and sample backup paths shown with dashed lines.

$BP(\ell_0^1) = \{\ell_0^2, \ell_1^2, \ell_1^1\}$	$BP(\ell_0^2) = \{\ell_0^3, \ell_1^3, \ell_1^2\}$	$BP(\ell_0^3) = \{\ell_0^1, \ell_1^1, \ell_1^3\}$
$BP(\ell_1^1) = \{\ell_0^2, \ell_1^2, \ell_0^3, \ell_1^3, \ell_3^1\}$	$BP(\ell_1^2) = \{\ell_0^3, \ell_1^3, \ell_0^1, \ell_1^1, \ell_2^2\}$	$BP(\ell_1^3) = \{\ell_0^1, \ell_1^1, \ell_0^2, \ell_1^2, \ell_2^3\}$
$BP(\ell_2^1) = \{\ell_2^2, \ell_3^2, \ell_3^1\}$	$BP(\ell_2^2) = \{\ell_2^3, \ell_3^3, \ell_3^2\}$	$BP(\ell_2^3) = \{\ell_2^1, \ell_3^1, \ell_3^3\}$
$BP(\ell_3^1) = \{\ell_2^2, \ell_3^2, \ell_0^3, \ell_1^3, \ell_0^1\}$	$BP(\ell_3^2) = \{\ell_2^3, \ell_3^3, \ell_0^1, \ell_1^1, \ell_0^2\}$	$BP(\ell_3^3) = \{\ell_2^1, \ell_3^1, \ell_0^2, \ell_1^2, \ell_0^3\}$

(c) List of all backup paths for the 2-resilient scheme on the 3-cube.

Figure 3 Illustration of $BP(.)$ on 2 and 3 dimensional cubes.

and $b' = (2^{d'-1})_2 \oplus b$. Let $\left(L_0^{d'}[\ell_i^d], L_1^{d'}[\ell_i^d]\right)$ denote the (unique) pair of incident d'-dim links, i.e. $L_0^{d'}[\ell_i^d] = (a, a')$ and $L_1^{d'}[\ell_i^d] = (b, b')$. The subscripts 0 and 1 indicate the value at the dth bit position of the links in the pair. Due to symmetry, $N_{d'}(N_{d'}(\ell_i^d)) = \ell_i^d$ and $L_b^{d'}[\ell_i^d] = L_b^{d'}[N_{d'}(\ell_i^d)], b \in \{0, 1\}$.

We formulate the backup path of a d-dim link as a set consisting of one d-dim link and pairs of links. These pairs constitute a joint path, i.e., two paths over the endpoints of detoured d-dim links. We refer to this joint path as a backup path and we always traverse it towards the included d-dim link. However in reality, a packet traverses the two paths in opposite directions, towards and away from the respective d-dim link.

For instance, the backup path of the first 1-dim link (i.e. ℓ_0^1) includes the 1-dim link reached via the incident pair of 2-dim links, and the pair itself (see Figure 3b): $BP(\ell_0^1) = \{L_0^2[\ell_0^1], L_1^2[\ell_0^1], N_2(\ell_0^1)\} = \{\ell_0^2, \ell_1^2, \ell_1^1\}$ (see Figure 3c). For the second 1-dim link we use the same pair, but we have to detour ℓ_0^1 in order to avoid conflict:

$$BP(\ell_1^1) = \{L_1^2[\ell_1^1], L_1^2[\ell_1^1], L_0^3[\ell_0^1], L_1^3[\ell_0^1], N_3(\ell_0^1)\} = \{\ell_0^2, \ell_1^2, \ell_0^3, \ell_1^3, \ell_3^1\}.$$

In general, the backup path of ℓ_i^d begins with the pair $\left(L_0^{d+1}[\ell_i^d], L_1^{d+1}[\ell_i^d]\right)$. If the first d-dim link, i.e. $N_{d+1}(\ell_i^d)$, is conflicting, then one continues by detouring this link via the pair of $(d+2)$-dim links and detours further d-dim links, until one reaches a d-dim link that is not conflicting, then traverses this link. Moreover, the jth detour is performed via the pair of $(d+j)$-dim links. Hence the pairs are traversed in the ascending order of consecutive dimensions. We denote the closure form of $N_d(.)$ w.r.t. this ordering as

$$N^{(j)}(\ell_i^d) := N_{d+j}(N_{d+j-1}(\dots N_{d+1}(\ell_i^d)\dots), 1 \le j < k.$$

We can now describe our backup path scheme formally, we refer to Figure 3c for an example listing all generated backup paths on the 3-dimensional hypercube. For each

dimension $d \in [k]$ and every $0 \le i < 2^{k-1}$, the backup path of ℓ_i^d is

$$\begin{aligned} BP(\ell_i^d) = \Big\{ & L_0^{d+1}[\ell_i^d], L_1^{d+1}[\ell_i^d], \\ & L_0^{d+2}[N_{d+1}(\ell_i^d)], L_1^{d+2}[N_{d+1}(\ell_i^d)], \\ & \dots, \\ & L_0^{d+r}[N^{(r-1)}(\ell_i^d)], L_1^{d+r}[N^{(r-1)}(\ell_i^d)], \\ & N^{(r)}(\ell_i^d)) = \ell_{i'}^d \Big\}. \end{aligned} \tag{1}$$

The path detours $r-1$ links, where r is the number of link pairs necessary to have, in order to reach the non-conflicting link $\ell_{i'}^d$ with smallest index. Therefore the path length is $2r+1$. We will later argue that $r \le R := \lceil \log k \rceil$.

Alternatively to the explicit formulation in (1), $\ell_{i'}^d$ can be obtained directly using bitwise operations. Assume $\ell_i^d = (a, b)$ and $\ell_{i'}^d = (a', b')$. By comparing a' to a (b' to b), we can see that only the r bits to the left of dth bit are affected, i.e., the bits $d+1$ to $d+R \pmod{k}$. For $x \in \{0,1\}^k$ and $s := R - (k-d)$, we define the increment function that determines the successor link as $inc_{s,d}(x) := (x^{>>s} + (2^d)_2^{>>s})^{>>-s}$. Here the $+$ ignores the carry flag out of the leftmost position. Therefore, $a' = inc_{s,d}(a)$ and $b' = inc_{s,d}(b)$. It is clear that the overall computation takes polynomial time.

In the next section, we will state some necessary observations regarding our hypercube construction, which we will employ for the correctness proof of our scheme in Section 3.3.

3.2 Proof Preliminaries

According to our backup path formulation (1), the backup path of a d-dim link passes through a d-dim link reached via links in higher dimensions, which are presented in pairs in (1). The backup path possibly detours some other d-dim links along its way. The pairs and the involved d-dim links together resemble a chain-like structure which facilitates describing some properties in this section. We now describe these structures formally.

▶ **Definition 3.** Given a *sequence of dimensions* $S_d := (d_i)_{i=0}, d_i \in [k] \setminus \{d\}$, a *chain* of d-dim links, starting from $\ell_{i_0}^d$, denoted by $C(\ell_{i_0}^d)$, consists of a subset of d-dim links and pairs of d_i-dim links, $d_i \in S_d$. The pairs form two walks over the endpoints of the contained d-dim links. The two parallel walks jointly *traverse* the chain. We denote the chain by $C_{S_d}(\ell_{i_0}^d) := \{\dots, \ell_{i_j}^d, (L_0^{d_j}[\ell_{i_j}^d], L_1^{d_j}[\ell_{i_j}^d]), \ell_{i_{j+1}}^d, \dots\}, j \ge 0, \ell_{i_{j+1}}^d = N_{d_j}(\ell_{i_j}^d)$. Moreover, if $\exists \ell_{i_{j'}} \in C(\ell_{i_0}^d) : \ell_{i_{j'+1}}^d = \ell_{i_0}$, then it is a *closed* chain denoted by C_{S_d}.

We can directly obtain the following property.

▶ **Property 4.** *Starting from any link ℓ_i^d, by traversing a chain $C_{S_d}(\ell_i^d)$, assume we arrive back at the same link. Then it must be the case that S_d contains every dimension an even number of times.*

▶ **Definition 5.** A link $(a, b), a < b$ is traversed in *uphill* direction when it is from a. The opposite is a *downhill* direction.

Based off this definition, we can categorize the traversal directions.

▶ **Property 6.** *Consider a closed chain containing the pair $(L_0^{d'}[\ell_i^d], L_1^{d'}[\ell_i^d])$ traversed between the links ℓ_i^d and $\ell_j^d = N_{d'}(\ell_i^d)$. If $j > i$ then the direction from ℓ_i^d to ℓ_j^d is uphill, otherwise downhill.*

▶ **Property 7.** *By Properties 4 and 6, in a closed chain, the number of traversals in every dimension is even, half of which is in downhill (uphill) direction.*

Intuitively, uphill and downhill traversals cancel each other which consequently turns the joint walks into joint closed walks over the endpoints.

We next study the interaction between chains. Let $S_d, S_{d'}, d' \neq d$ be two sequences of dimensions. We say the chain $C_{S'_d}$ *crosses* the chain C_{S_d} if $\exists P := (L^0_d, L^1_d) \in C_{S_{d'}} : P \cap C_{S_d} \neq \emptyset$. That is, $C_{S'_d}$ traverses a pair of d-dim links, at least one of which belongs to C_{S_d}.

▶ **Definition 8.** A *mixed* chain is the concatenation of multiple chains (over several dimensions) that cross each other consecutively. In other words, a mixed chain consists of chains of links in at least two dimensions. Formally, for given dimensions $d, d', d'' \in [k]$ and sequences S_d and $S_{d'}$, assume the chain $C_{S_{d'}} = \{\dots, \ell^{d'}_x, (L^0_d[\ell^{d'}_x] = \ell^d_y, L^1_d[\ell^{d'}_x]), \dots\}$ crosses $C_{S_d} = \{\dots, \ell^d_y, (L^0_{d''}[\ell^d_y], L^1_{d''}[\ell^d_y]), \dots\}$. We concatenate these chains into a mixed chain as $\{\dots, \ell^{d'}_x, \ell^d_y, (L^0_{d''}[\ell^d_y], L^1_{d''}[\ell^d_y]), \dots\}$.

The observations in the Properties (4), (6), and (7) hold for mixed chains as well. This is because the mentioned properties do not depend on the dimension of the links being chained, but only on dimensions that are actually traversed. However, traversing a chain of d-dim links does not always imply that dimension d is traversed. Consider three chains of d, d', and d''-dim links that cross each other consecutively. E.g., the chain $C_{S_d} = \{\dots, \ell^d_{j_L}, \dots, \ell^d_{j_R}, (L^0_{d''} = \ell^{d''}_{j_R+1}, L^1_{d''}), \dots\}$ that is crossed by the chain $C_{S_{d'}} = \{\dots, \ell^{d'}_{j_L-1}, (L^0_d = \ell^d_{j_L}, L^1_d), \dots\}$ at the link $\ell^d_{i_{j_L}}$. Also, C_{S_d} crosses the chain $C_{S_{d''}} = \{\dots, \ell^{d''}_{i_{j_{R+1}}}, \dots\}$ at the link $\ell^{d''}_{i_{j_R+1}}$. We examine whether dimension d is traversed by comparing the dth bit of the last link before the first cross to C_{S_d}, i.e. $\ell^{d'}_{i_{j_L-1}} = (a_0, b_0)$, to the dth bit of the first link after the second cross (by C_{S_d}), i.e. $\ell^{d''}_{i_{j_R+1}} = (a_1, b_1)$. Dimension d is traversed if and only if the two bits hold different values. That is, $(a_0 \wedge a_1) \wedge (2^{d-1})_2 = 0$.

A backup path $BP(\ell^d_i) = \{(L^0_{d+1}, L^1_{d+1}), \dots, N^*(\ell^d_i)\}$ can be represented as a chain $C(\ell^d_i) := \{\ell^d_i, \{L^0_{d+1}, L^1_{d+1}\}, N^{(1)}(\ell^d_i), \dots, N^{(*)}(\ell^d_i)\}$. By Definition 2, there is a dependency from ℓ^d_i to every other link $\ell^{d'}_{i'} \in BP(\ell^d_i)$. Let $MC(\ell^d_i, \ell^{d'}_{i'}) \subseteq C(\ell^d_i) \cup \{\ell^{d'}_{i'}\}$ denote the mixed chain up to and including $\ell^{d'}_{i'}$. Consider the set of backup paths of some subset of links $\{\ell^{d_0}_{i_0}, \ell^{d_1}_{i_1}, \dots, \ell^{d_{x-1}}_{i_{x-1}}\}$ that induce a cycle of dependencies. Each dependency corresponds to a mixed chain, concatenating them sequentially, yields the closed mixed chain $\mathcal{MC} := MC(\ell^{d_0}_{i_0}, \ell^{d_1}_{i_1}) \cup MC(\ell^{d_1}_{i_1}, \ell^{d_2}_{i_2}) \cup \dots \cup MC(\ell^{d_{x-1}}_{i_{x-1}}, \ell^{d_0}_{i_0})$. Recall that in $BP(.)$, pairs connecting consecutive same-dimension links are traversed in the ascending order of dimensions. Therefore, the sequence of dimensions traversed by $\mathcal{MC}$ is specified by $(\tilde{d}_i)_{i=0}$, where $\tilde{d}_0 = 0$, and either $\tilde{d}_{j+1} = \tilde{d}_j$ or $\tilde{d}_{j+1} = \tilde{d}_j + 1 \pmod{k}$. From now on, we assume only the closed chains restricted to the sequence of dimensions $\tilde{d}_i$.

3.3 Correctness

In the following, we address the correctness of our backup path scheme, i.e., resilience to up to $k-1$ link failures in k-dimensional hypercubes. To this end, we need one additional result:

▶ **Claim 9.** *In any backup path $p := BP(\ell^d_i)$ at most one pair of links is traversed in uphill direction.*

Proof. If p does not detour any link then the only pair of links, i.e. $(L^0_{d+1}(\ell^d_i)), L^1_{d+1}(\ell^d_i)))$, is traversed either in uphill or downhill direction, which trivially satisfies the claim. If p detours some link ℓ^d_j, then $j < i$ (by construction). By Property 6, the pair of links preceding

ℓ_j^d is traversed in downhill direction. Since p does not detour the last d-dim link, only the last pair (preceding the last link) is possibly traversed in uphill direction. ◀

We can now prove our main result:

▶ **Theorem 10.** *The scheme $BP(.)$ listed in Term (1) is $(k-1)$-resilient.*

Proof. In order to show that the scheme is $(k-1)$-resilient, we argue that any cycle of dependencies consists of at least k links. We first show that for every $d \in [k]$, any cycle of dependencies over d-dim links is of length at least k. The backup path of every link ℓ_i^d uses only one d-dim link $\ell_{i'}^d, i' = i+1 \pmod{R}$. Hence, the set of d-dim links are dependent sequentially. Therefore, having $R = \lceil \log k \rceil$ is sufficient to ensure any cycle of dependencies induced by d-dim links is of length $2^R \geq k$.

It remains to analyze the dependency cycles that consist of links in multiple dimensions. By Definition 8 and the construction of the $\mathcal{MC}$, such cycles correspond to mixed chains in the k-cube, each having the following properties:

1. Due to the non-descending sequence $\tilde{d}_i$ and by Property 4, $\mathcal{MC}$ traverses the sequence of dimensions $1, \ldots k$ an even number of times, therefore there are at least $2k$ traversals.
2. By Property 7, at least k of the traversals are in uphill direction.
3. By Property 9, a backup path takes at most one uphill. Meaning, each dependency contributes at most one uphill traversal to the mixed chain.

Combining (1), (2), and (3), implies that there must be at least k dependencies in the assumed cycle of dependencies, which concludes our claim. ◀

4 Related Work

Most modern communication networks support some form of resilient routing, and the topic has already received much interest in the literature. There exists much literature on single [16, 47, 65, 68], double [12, 49], and more [15] failure scenarios, the latter being motivated by, e.g., shared risk link groups [57], attacks [61], or simply node failures which affect all incident links [3, 15, 28, 56]. The spectrum of solutions is broad as well, with some solutions providing only heuristic guarantees [12, 49], some schemes exploiting packet-header rewriting [8, 15] (which however is not always supported in existing networks) or packet-duplication [32] (which however comes with overheads). Furthermore, there is also work that aims at quickly optimizing network behavior after link failures have propagated, e.g., by pre-computing how to rescale traffic at ingress routers once these nodes are fault-aware [43]. However, such mechanisms do not provide protection for packets during convergence.

An interesting line of research studies mechanisms which do not require any additional information in the packet header, such as the works by Feigenbaum et al. [17], by Chiesa et al. [10, 11] (establishing an interesting connection to arc-disjoint graph covers), by Elhourani et al. [15], by Stephens et al. [59, 60], by Borokhovich et al. [6], by Pignolet et al. [51] (establishing an interesting connection to distributed computing problems without communication [45]), and by Foerster et al. [23]. However, these solutions do not require failover paths to traverse the nodes of the original path and do not account for the specific properties of the networks considered in this paper. The former is particularly motivated by the advent of (virtualized [18]) middleboxes [9], and is also known as *local protection scheme* in MPLS terminology [55].

Our work is situated in the context of MPLS and Segment Routing (SR) networks where routing is based on stacks and more specifically, the top of the stack label [50]. While the

design of resilient routing algorithms has received much attention already in the context of MPLS, see e.g., [31] and [55, 36] and references therein, existing research on SR networks mainly revolves around flow control, traffic engineering and network utilization [5, 63, 13, 42], or network monitoring [4], see the works by Filsfils et al. [21] and Lebrun et al. [14, 38, 41, 40] for a good overview. Optimization problems typically include the minimization of the number of segments required to compute segmented paths [29]. Salsano et al. [54] propose methods to leverage SR in a network without requiring extensions to routing protocols, and Hartert et al. [34] propose a framework to express and implement network requirements in SR. Only little is known today about fast rerouting in SR networks. In [22], it has been shown that existing solutions for SR fast failover, based on TI-LFA [25], do not work in the presence of two or more failures. However, [22] relies on failure-carrying packets, which is undesirable as discussed above and we overcome in the current paper. Finaly, we in this paper considered hypercubes, which have recently been studied for local fast failover algorithms in [11, 24] as well. While for a single link failure, the general approach of François et al. [25] can be used, we are not aware of any approaches that (conceptually) employ Segment Routing for local fast failover in hypercubes for multiple failures.

5 Conclusion and Future Work

This paper studied the design of algorithms for local fast failover in Segment Routing networks, subject to multiple link failures. Our main result is a maximally robust, $(k-1)$-resilient algorithm for k-dimensional hypercubes, which can be computed efficiently.

We see our work as a first step and believe that it opens several promising directions for future research. On the algorithmic side, it would be interesting to extend the study to algorithms for other graph classes, also providing a minimal number of segments or requiring a minimal number of forwarding rules. On the practical side, given that segment routing is ready to be deployed in IPv6 environments, it would be interesting to study experimental evaluations, which can in turn also refine our model. In the following, we provide some first directions.

5.1 Future Work I: Resilient Segment Routing on General Graphs

It will be interesting to study the complexity of fast rerouting on general graphs, and develop (approximation) algorithms accordingly. We conjecture that computing backup path schemes with maximal resiliency is NP-hard on general graphs. In non-polynomial time, a Mixed Integer Program (MIP) formulation can provide an optimal solution for general graphs. The following MIP considers the problem of generating a small number of required segments for the backup paths, and if the desired resiliency cannot be met, at least maximizes the number of protected links. We hope that our MIP formulation can aid the community in developing further backup path schemes, e.g., by using it as a baseline comparison to evaluate the quality of polynomial runtime algorithms for different graph classes beyond the hypercube.

More specifically, the MIP presented next will compute an f-resilient backup path allocation that is optimal in the number of protected links. For completeness purposes, we consider directed graphs $G=(V,E)$. As our MIP is also concerned with the number of labels for each backup path, we provide some additional preliminaries relevant to practical implementations. A backup path in general can be subdivided into path segments, each being a shortest path between its endpoints: such a path segment will only need one label on the stack, when the nodes employ shortest path routing. However, when the network utilizes link weights, some backup paths cannot be represented by node labels [25]: e.g., if a link on

the backup path has infinite weight, while all other links have unit weight. For these corner cases, we need to allow single links as items on the label stack, which we denote as *tunnel* links: In the worst case, the whole backup path contains only tunnel links. Should a tunnel link physically fail, the corresponding label will be popped to prevent stuck packets (a failed link cannot be traversed), and the respective backup path will be traversed.

$$\textbf{Maximize} \sum_{\ell \in E} \mathcal{I}_\ell \tag{2}$$

$$\mathcal{SP}^z_\ell = \begin{cases} 1 & \ell \in SP(u,z) \\ 0 & else \end{cases} \qquad \forall \ell = (u,v) \in E, z \in V \tag{3}$$

$$\mathcal{D}_{\ell\ell'}, \mathcal{X}^v_{\ell\ell'}, \mathcal{T}_{\ell\ell'}, \mathcal{W}^v_\ell, \mathcal{I}_\ell, \in \{0,1\} \qquad \forall \ell, \ell' \in E, \forall v \in V \tag{4}$$

$$\mathcal{D}_{\ell\ell} = 0 \qquad \forall \ell \in E \tag{5}$$

$$\sum_{\ell_2=(v,*)} \mathcal{D}_{\ell_1\ell_2} - \sum_{\ell_2=(*,v)} \mathcal{D}_{\ell_1\ell_2} = \begin{cases} \mathcal{I}_{\ell_1} & v = s \\ -\mathcal{I}_{\ell_1} & v = t \\ 0 & \text{else} \end{cases} \qquad \forall \ell_1 = (s,t) \in E, v \in V \tag{6}$$

$$\mathcal{X}^v_{\ell_1\ell_2} \le \mathcal{SP}^v_{\ell_2}, \sum_{\ell \in E, \ell \ni v} \mathcal{D}_{\ell_1\ell} \qquad \forall \ell_1, \ell_2 \in E, v \in V \tag{7}$$

$$\mathcal{D}_{\ell_1\ell_2} \le \mathcal{SP}^t_{\ell_2} + \mathcal{T}_{\ell_1\ell_2} + \sum_{v \in V} \mathcal{X}^v_{\ell_1\ell_2} \qquad \forall \ell_1 = (s,t), \ell_2 \in E \tag{8}$$

$$d_{\ell_1\ell_2} \ge 0, d_{\ell_1\ell_1} = 0 \qquad \forall \ell_1, \ell_2 \in E \tag{9}$$

$$d_{\ell_1\ell_3} \le d_{\ell_1\ell_2} + 1 + (1 - \mathcal{D}_{\ell_2\ell_3}) \times \infty \qquad \forall \ell_1, \ell_2, \ell_3 \in E \tag{10}$$

$$d_{\ell_1\ell_2} + d_{\ell_2\ell_1} \ge f + 1 \qquad \forall \ell_1, \ell_2 \in E \tag{11}$$

$$\mathcal{W}^v_{\ell_1} \ge \mathcal{X}^v_{\ell_1\ell_2} \qquad \forall \ell_1\ell_2 \in E, v \in V \tag{12}$$

$$\sum_{v \in V} \mathcal{W}^v_\ell + \sum_{\ell' \in E} \mathcal{T}_{\ell\ell'} \le \text{LABELS} \qquad \forall \ell \in E \tag{13}$$

Armed with the above preliminaries, we can now provide a general overview of the MIP. Let $SP(u,z)$ be the shortest path between u and z.[4] For every link $\ell = (s,t)$, we pre-compute constants $\mathcal{SP}^z_\ell$, each indicating whether the shortest path from s to z includes ℓ or not. With respect to the logical flow of the formulation, the MIP first computes a backup path $\mathcal{P}_\ell = \{\ell' \in E \mid \mathcal{D}_{\ell\ell'} = 1\}$ for every link $\ell \in E$. Then, for every link $\ell' \in \mathcal{P}_\ell$ whose shortest path to t does not take the link itself (i.e. $\mathcal{SP}^t_{\ell'} = 0$), the MIP either finds an intermediate node v such that $\mathcal{SP}^v_{\ell'} = 1$, or flags the link as a tunnel link (with $\mathcal{T}_{\ell_1\ell_2}$). As a result, every link of $\mathcal{P}_\ell$ either is a tunnel link or is on the shortest path to a next intermediate node, if not t (i.e. on a segment). This is imposed by the set of constraints (7) and (8). With constraints (9) to (11), we ensure an f-resilient backup path selection. Constraint (11) forbids any cyclic dependency of length $\le f$. At the end, the MIP restricts the number of segments to the constant LABELS.

Next, we explain each set of constraints and variables more technically.

- (2): maximizing the number of protected links. The failure of any subset of up to f protected links can be tolerated.
- (3): are the pre-computed shortest path trees for all nodes.

[4] Should there be multiple options for shortest paths, we pick them in such a way that each subpath of a shortest path is again a shortest path.

- (4): each variable $\mathcal{D}_{\ell\ell'}$ is set to 1 if ℓ' is designated to the backup path of ℓ ($\mathcal{P}_\ell$), otherwise remains 0. Each variable $\mathcal{X}^v_{\ell\ell'}$ indicates whether 1) the node v is a waypoint on $\mathcal{P}_\ell$ and 2) ℓ' is on the shortest path from the tail of ℓ' to v, hence on the backup path. Similarly, $\mathcal{T}_{\ell\ell'}$ indicates whether ℓ' is a tunnel link on $\mathcal{P}_\ell$. Variables $\mathcal{W}^v_\ell$ is set to 1 when some node v is used as a waypoint for $\mathcal{P}_\ell$. Each variable $\mathcal{I}_\ell$ indicated whether ℓ is protected.
- (6): these constraints enforce the links specified by $\mathcal{D}_{\ell_1 *}$ to form a simple path connecting the endpoints of ℓ_1, not using ℓ_1 (due to (5)).
- (7), (8): a link $\ell_2 = (x, y)$ is allowed to be on the the backup path $\mathcal{P}_{\ell_1}, \ell_1 = (s, t)$ only if
  1. the link ℓ_2 is on the shortest path $SP(x, t)$ i.e. $SP^t_{\ell_2} = 1$;
  2. else, a node $v \in \mathcal{P}_{\ell_1}$ exists s.t. $SP(x, v)$ begins with ℓ_2 (when $\mathcal{X}^v_{\ell_1\ell_2} = 1$ in (7)),
  3. else, the variable $\mathcal{T}_{\ell_1\ell_2}$ is set to 1, which enforces the link ℓ_2 on $\mathcal{P}_{\ell_1}$ as a tunnel link.

  Therefore at least one of the cases must apply to the pair ℓ_1, ℓ_2 in order to have $\mathcal{D}_{\ell_1\ell_2} = 1$ feasible. Cases 2 and 3 correspond to adding new segments. Note that the case 3 can trivially hold for any link which would result in unrestricted number of segments. But latter constraints avoid this in favour of having fewer segments.
- (9),(10),(11): here we formulate the all-pairs shortest path sub-problem on the dependency graph induced by $\mathcal{D}_{**}$. Given a feasible assignment, the value of each d_{xy} is at most the length of the shortest path from x to y. The length of the shortest cycle of dependencies through each dependency arc (ℓ_1, ℓ_2) is constrained by (11).
- (12),(13): the flag $\mathcal{W}^v_\ell$ is set to 1 whenever the node $v \in \mathcal{P}_\ell$ is used as a waypoint for some ℓ' on $\mathcal{P}_\ell$. We restrict the total number of labels (thus, the number of segments) using the constant LABELS.

5.2 Future Work II: Testbeds for Fast Failover in Segment Routing

The most popular testing environment for Segment Routing is *Nanonet* [39], which provides an IPv6 data plane and is conceptually based off *Mininet*[5]. Nanonet allows to easily benchmark Segment Routing in different topologies, all contained in a virtualized enviroment.

To conduct a first feasibility study and evaluate the performance of Segment Routing under different failure scenarios, we deploy the example from Figure 2 as a topology, with an additional source node s connected to v_1, using w as the destination node. Each link has 1 ms delay and bidirectional 10 Mbit/s bandwidth. Without failures, the standard route is s—v_1–(e_1)–w.

If e_1 is unavailable, then v_1 will push v_2 as a segment label (and w will switch to e_2 for the return path), i.e., the packet path is s—v_1—v_2–(e_2)–w. When additionally e_2 is unavailable, then v_1 will push v_1, v_3 as segment labels (with w switching to e_3 for the return path), with the total packet path being s—v_1—v_2—v_1—v_3–(e_3)–w.

We use `iperf3` to generate IPv6 traffic to evaluate the TCP throughput between source and destination nodes, stopping the experiment after 20 seconds, providing ample time for TCP to stabilize. As Nanonet does not support failing links during runtime, we run the experiment three times, first without link failures, then deactivating e_1, and lastly deactivating e_1 and e_2. The results of all three experiments are plotted in Figure 4.

As can be seen, the throughput slightly deteriorates after one link failure, with an additional very small performance hit after the second link failure. We believe that the extent of the slowdown may be related to simulation constraints, as implementing Segment Routing takes additional computational overhead in the virtualized environment, but it would be

[5] `http://mininet.org/`

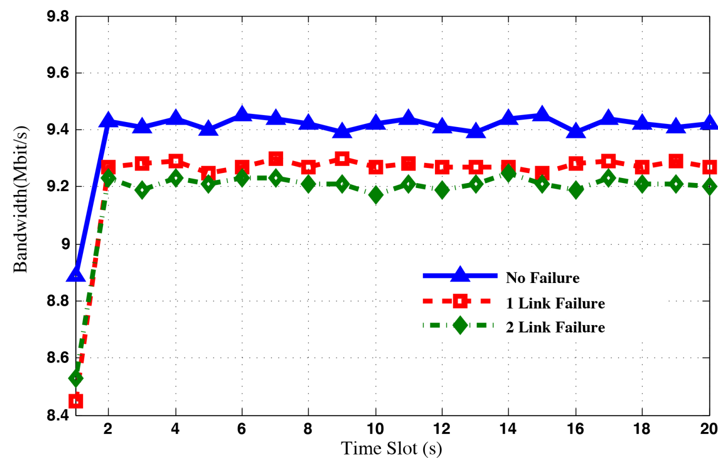


Figure 4 TCP throughput of `iperf3` under 0, 1, and 2 link failures in Nanonet, using an adapted version of the topology and Segment Routing rules from Figure 2.

interesting to investigate the performance impact in a real hardware testbed. Additionally, we believe it would be worthwhile to implement link failures during the simulation runtime in Nanonet, to efficiently estimate the possible performance changes that occur directly after the links went down. We plan to extend our current simulations in these directions.

References

1. Saeed Akhoondian Amiri, Klaus-Tycho Foerster, Riko Jacob, Mahmoud Parham, and Stefan Schmid. Waypoint Routing in Special Networks. In *Proc. IFIP Networking*, 2018.
2. Saeed Akhoondian Amiri, Klaus-Tycho Foerster, Riko Jacob, and Stefan Schmid. Charting the Algorithmic Complexity of Waypoint Routing. *CCR*, 48(1):42–48, 2018. `doi:10.1145/3211852.3211859`.
3. Alia K Atlas and Alex Zinin. Basic specification for IP fast-reroute: loop-free alternates. *IETF RFC 5286*, 2008.
4. François Aubry, David Lebrun, Stefano Vissicchio, Minh Thanh Khong, Yves Deville, and Olivier Bonaventure. Scmon: Leveraging segment routing to improve network monitoring. In *Proc. IEEE INFOCOM*, 2016.
5. Randeep Bhatia, Fang Hao, Murali Kodialam, and TV Lakshman. Optimized network traffic engineering using segment routing. In *IEEE INFOCOM*, 2015.
6. Michael Borokhovich and Stefan Schmid. How (Not) to Shoot in Your Foot with SDN Local Fast Failover: A Load-Connectivity Tradeoff. In *OPODIS*, 2013.
7. Costas Busch, Srikanth Surapaneni, and Srikanta Tirthapura. Analysis of link reversal routing algorithms for mobile ad hoc networks. In *Proc. ACM SPAA*. ACM, 2003. `doi:10.1145/777412.777446`.
8. Marco Canini, Petr Kuznetsov, Dan Levin, and Stefan Schmid. A Distributed and Robust SDN Control Plane for Transactional Network Updates. In *Proc. IEEE INFOCOM*, 2015.
9. B. Carpenter and S. Brim. Middleboxes: Taxonomy and Issues. RFC 3234, RFC Editor, February 2002. URL: `http://www.rfc-editor.org/rfc/rfc3234.txt`.
10. 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. `doi:10.1109/TNET.2016.2619398`.

11 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. doi:10.1109/INFOCOM.2016.7524552.

12 Hongsik Choi, Suresh Subramaniam, and Hyeong-Ah Choi. On double-link failure recovery in WDM optical networks. In *Proc. IEEE INFOCOM*, 2002. doi:10.1109/INFCOM.2002.1019327.

13 Luca Davoli, Luca Veltri, Pier Luigi Ventre, Giuseppe Siracusano, and Stefano Salsano. Traffic engineering with segment routing: SDN-based architectural design and open source implementation. In *Proc. EWSDN*, 2015.

14 Fabien Duchêne, David Lebrun, and Olivier Bonaventure. SRv6Pipes: enabling in-network bytestream functions. In *Proc. IFIP Networking*, 2018.

15 Theodore Elhourani, Abishek Gopalan, and Srinivasan Ramasubramanian. IP fast rerouting for multi-link failures. *IEEE/ACM Trans. Netw*, 24(5):3014–3025, 2016.

16 Gábor Enyedi, Gábor Rétvári, and Tibor Cinkler. A novel loop-free IP fast reroute algorithm. In *Meeting of the European Network of Universities and Companies in Information and Communication Engineering*, pages 111–119. Springer, 2007.

17 Joan Feigenbaum et al. BA: On the resilience of routing tables. In *Proc. ACM PODC*, 2012.

18 ETSI. Network Functions Virtualisation. In *White Paper*, 2013.

19 Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to SDN: an intellectual history of programmable networks. *ACM SIGCOMM CCR*, 44(2):87–98, 2014.

20 Clarence Filsfils, Pierre François, Stefano Previdi, Bruno Decraene, Stephane Litkowski, Martin Horneffer, Igor Milojevic, Rob Shakir, Saku Ytti, Wim Henderickx, Jeff Tantsura, Sriganesh Kini, and Edward Crabbe. Segment Routing Architecture. In *Segment Routing Use Cases, IETF Internet-Draft*, 2014.

21 Clarence Filsfils, Nagendra Kumar Nainar, Carlos Pignataro, Juan Camilo Cardona, and Pierre Francois. The segment routing architecture. In *IEEE GLOBECOM*, 2015.

22 Klaus-Tycho Foerster, Mahmoud Parham, Marco Chiesa, and Stefan Schmid. TI-MFA: keep calm and reroute segments fast. In *Global Internet Symposium (GI)*, 2018.

23 Klaus-Tycho Foerster, Yvonne-Anne Pignolet, Stefan Schmid, and Gilles Tredan. Local Fast Failover Routing With Low Stretch. *ACM SIGCOMM CCR*, 1:35–41, January 2018.

24 Klaus-Tycho Foerster, Yvonne Anne Pignolet, Stefan Schmid, and Gilles Trédan. Local Fast Failover Routing With Low Stretch. *CCR*, 48(1):35–41, 2018. doi:10.1145/3211852.3211858.

25 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.

26 Pierre François, Clarence Filsfils, John Evans, and Olivier Bonaventure. Achieving subsecond IGP convergence in large IP networks. *CCR*, 35(3):35–44, 2005. doi:10.1145/1070873.1070877.

27 Eli M. Gafni and Dimitri P. Bertsekas. Distributed Algorithms for Generating Loop-Free Routes in Networks with Frequently Changing Topology. *IEEE Transactions on Communications*, 29(1):11–18, January 1981.

28 Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *ACM SIGCOMM CCR*, volume 41, pages 350–361, 2011.

29 Alessio Giorgetti, Piero Castoldi, Filippo Cugini, Jeroen Nijhof, Francesco Lazzeri, and Gianmarco Bruno. Path encoding in segment routing. In *Proc. IEEE GLOBECOM*, 2015.

30 Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. BCube: a high performance, server-centric network architecture for modular data centers. In *Proc. ACM SIGCOMM*, 2009.

31 Anupam Gupta, Amit Kumar, and Rajeev Rastogi. Traveling with a pez dispenser (or, routing issues in mpls). *SIAM Journal on Computing*, 34(2):453–474, 2005.

32 Prashanth Hande, Mung Chiang, Robert Calderbank, and Sundeep Rangan. Network pricing and rate allocation with content-provider participation. In *Proc. IEEE INFOCOM*, 2010.

33 Ed Harrison, Adrian Farrel, and Ben Miller. Protection and restoration in MPLS networks. *Data Connection White Paper*, 2001.

34 Renaud Hartert, Stefano Vissicchio, Pierre Schaus, Olivier Bonaventure, Clarence Filsfils, Thomas Telkamp, and Pierre Francois. A declarative and expressive approach to control forwarding paths in carrier-grade networks. In *ACM SIGCOMM CCR*, volume 45, pages 15–28, 2015.

35 Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven WAN. In *Proc. ACM SIGCOMM*, 2013.

36 Jesper Stenbjerg Jensen, Troels Beck Krogh, Jonas Sand Madsen, Stefan Schmid, Jiri Srba, and Marc Tom Thorgersen. P-Rex: Fast Verification of MPLS Networks with Multiple Link Failures. In *Proc. ACM CoNEXT*, 2018.

37 Karthik Lakshminarayanan, Matthew Caesar, Murali Rangan, Tom Anderson, Scott Shenker, and Ion Stoica. Achieving convergence-free routing using failure-carrying packets. In *Proceedings of the ACM SIGCOMM 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Kyoto, Japan, August 27-31, 2007*, pages 241–252. ACM, 2007. doi:10.1145/1282380.1282408.

38 David Lebrun. *Reaping the Benefits of IPv6 Segment Routing*. PhD thesis, UCLouvain / ICTEAM / EPL, October 2017.

39 David Lebrun. Virtual networks testing framework (Nanonet). https://github.com/segment-routing/nanonet, February 2017.

40 David Lebrun and Olivier Bonaventure. Implementing IPv6 Segment Routing in the Linux Kernel. In *Proc. ACM ANRW*, 2017.

41 David Lebrun, Mathieu Jadin, François Clad, Clarence Filsfils, and Olivier Bonaventure. Software Resolved Networks: Rethinking Enterprise Networks with IPv6 Segment Routing. In *Proc. ACM SOSR*, 2018.

42 Ming-Chieh Lee and Jang-Ping Sheu. An Efficient Routing Algorithm Based on Segment Routing in Software-defined Networking. *Comput. Netw.*, 103(C):44–55, July 2016.

43 Hongqiang Harry Liu, Srikanth Kandula, Ratul Mahajan, Ming Zhang, and David Gelernter. Traffic engineering with forward fault correction. In *Proc. ACM SIGCOMM*, 2014.

44 Junda Liu, Aurojit Panda, Ankit Singla, Brighten Godfrey, Michael Schapira, and Scott Shenker. Ensuring Connectivity via Data Plane Mechanisms. In *Proc. USENIX NSDI*, 2013.

45 Grzegorz Malewicz, Alexander Russell, and Alexander A. Shvartsman. Distributed scheduling for disconnected cooperation. *Distributed Computing*, 18(6):409–420, 2005.

46 Athina Markopoulou, Gianluca Iannaccone, Supratik Bhattacharyya, Chen-Nee Chuah, Yashar Ganjali, and Christophe Diot. Characterization of failures in an operational IP backbone network. *IEEE/ACM Trans. Netw.*, 16(4):749–762, 2008. doi:10.1145/1453698.1453699.

47 Srihari Nelakuditi, Sanghwan Lee, Yinzhe Yu, Zhi-Li Zhang, and Chen-Nee Chuah. Fast local rerouting for handling transient link failures. *IEEE/ACM Trans. Netw*, 15(2):359–372, 2007.

48 Mohammad Noormohammadpour and Cauligi S Raghavendra. Datacenter Traffic Control: Understanding Techniques and Tradeoffs. *IEEE Communications Surveys & Tutorials*, 20(2):1492–1525, 2017.

49 Eunseuk Oh, Hongsik Choi, and Jong-Seok Kim. Double-Link Failure Recovery in WDM Optical Torus Networks. In *Information Networking, Networking Technologies for Broadband and Mobile Networks, International Conference ICOIN*, 2004.

50 P. Pan, G. Swallow, and A. Atlas. Fast Reroute Extensions to RSVP-TE for LSP Tunnels. RFC 4090, RFC Editor, May 2005.

51 Yvonne-Anne Pignolet, Stefan Schmid, and Gilles Tredan. Load-Optimal Local Fast Rerouting for Dependable Networks. In *Proc. IEEE/IFIP DSN*, 2017.

52 Y. Saad and M. H. Schultz. Topological properties of hypercubes. *IEEE Transactions on Computers*, 37(7):867–872, July 1988.

53 Yousef Saad and Martin H. Schultz. Data Communication in Hypercubes. *J. Parallel Distrib. Comput.*, 6(1):115–135, 1989. `doi:10.1016/0743-7315(89)90045-2`.

54 Stefano Salsano, Luca Veltri, Luca Davoli, Pier Luigi Ventre, and Giuseppe Siracusano. PMSR—Poor Man's Segment Routing, a minimalistic approach to Segment Routing and a Traffic Engineering use case. In *Proc. IEEE/IFIP NOMS*, 2016.

55 Stefan Schmid and Jiri Srba. Polynomial-Time What-If Analysis for Prefix-Manipulating MPLS Networks. In *Proc. IEEE INFOCOM*, 2018.

56 Aman Shaikh, Chris Isett, Albert Greenberg, Matthew Roughan, and Joel Gottlieb. A case study of OSPF behavior in a large enterprise network. In *Proc. ACM SIGCOMM Workshop on Internet Measurment*, 2002.

57 Lu Shen, Xi Yang, and Byrav Ramamurthy. Shared risk link group (SRLG)-diverse path provisioning under hybrid service level agreements in wavelength-routed optical mesh networks. *IEEE/ACM Transactions on Networking (ToN)*, 13(4):918–931, 2005.

58 Abhinav Kumar Singh, Ravindra Singh, and Bikash C. Pal. Stability Analysis of Networked Control in Smart Grids. *IEEE Trans. Smart Grid*, 6(1):381–390, 2015. `doi:10.1109/TSG.2014.2314494`.

59 Brent Stephens, Alan L. Cox, and Scott Rixner. Plinko: Building Provably Resilient Forwarding Tables. In *Proc. 12th ACM HotNets*, 2013.

60 Brent Stephens, Alan L Cox, and Scott Rixner. Scalable Multi-Failure Fast Failover via Forwarding Table Compression. *SOSR. ACM*, 2016.

61 János Tapolcai, Balázs Vass, Zalán Heszberger, József Bıró, David Hay, Fernando A Kuipers, and Lajos Rónyai. A Tractable Stochastic Model of Correlated Link Failures Caused by Disasters. In *Proc. IEEE INFOCOM*, 2018.

62 Frederic Trate. Bringing Segment Routing and IPv6 together, August 2016. URL: `https://blogs.cisco.com/sp/bringing-segment-routing-and-ipv6-together`.

63 George Trimponias, Yan Xiao, Hong Xu, Xiaorui Wu, and Yanhui Geng. On Traffic Engineering with Segment Routing in SDN based WANs. *arXiv preprint arXiv:1703.05907*, 2017.

64 Emmanouel A. Varvarigos and Dimitri P. Bertsekas. Performance of hypercube routing schemes with or without buffering. *IEEE/ACM Trans. Netw.*, 2(3):299–311, 1994. `doi:10.1109/90.311628`.

65 Junling Wang and Srihari Nelakuditi. IP fast reroute with failure inferencing. In *Proc. SIGCOMM Workshop on Internet Network Management*, pages 268–273, 2007.

66 Ye Wang, Hao Wang, Ajay Mahimkar, Richard Alimi, Yin Zhang, Lili Qiu, and Yang Richard Yang. R3: resilient routing reconfiguration. In *Proc. ACM SIGCOMM*, 2010.

67 Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Antony I. T. Rowstron. Better never than late: meeting deadlines in datacenter networks. In *Proc. ACM SIGCOMM*, 2011.

68 Baobao Zhang, Jianping Wu, and Jun Bi. RPFP: IP fast reroute with providing complete protection and without using tunnels. In *Proc. IWQoS*, 2013.

Effects of Topology Knowledge and Relay Depth on Asynchronous Appoximate Consensus

Dimitris Sakavalas
Boston College, USA
dimitris.sakavalas@bc.edu

Lewis Tseng
Boston College, USA
lewis.tseng@bc.edu

Nitin H. Vaidya[1]
Georgetown University, USA
nitin.vaidya@georgetown.edu

Abstract

Consider a point-to-point message-passing network. We are interested in the *asynchronous* crash-tolerant consensus problem in *incomplete networks*. We study the feasibility and efficiency of approximate consensus under different restrictions on topology knowledge and the *relay depth*, i.e., the maximum number of hops any message can be relayed. These two constraints are common in large-scale networks, and are used to avoid memory overload and network congestion respectively. Specifically, for positive integer values k and k', we consider that each node knows all its neighbors of at most k-hop distance (*k-hop topology knowledge*), and the relay depth is k'. We consider both *directed* and *undirected* graphs. More concretely, we answer the following question in asynchronous systems:

> *What is a tight condition on the underlying communication graphs for achieving approximate consensus if each node has only a k-hop topology knowledge and relay depth k'?*

To prove that the necessary conditions presented in the paper are also sufficient, we have developed algorithms that achieve consensus in graphs satisfying those conditions:

- The first class of algorithms requires k-hop topology knowledge and relay depth k. Unlike prior algorithms, these algorithms *do not* flood the network, and each node *does not* need the full topology knowledge. We show how the convergence time and the message complexity of those algorithms is affected by k, providing the respective upper bounds.
- The second set of algorithms requires only one-hop neighborhood knowledge, i.e., immediate incoming and outgoing neighbors, but needs to flood the network (i.e., relay depth is n, where n is the number of nodes). One result that may be of independent interest is a *topology discovery* mechanism to learn and "estimate" the topology in asynchronous directed networks with crash faults.

2012 ACM Subject Classification Computer systems organization → Fault-tolerant network topologies

Keywords and phrases Asynchrony, crash, consensus, incomplete graphs, topology knowledge

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.14

Related Version A full version of the paper is available at [21], https://arxiv.org/abs/1803.04513.

[1] This research is supported in part by National Science Foundation awards 1421918. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of the funding agencies or the U.S. government.

© Dimitris Sakavalas, Lewis Tseng, and Nitin H. Vaidya;
licensed under Creative Commons License CC-BY

Figure 1 Effect of increased k-hop knowledge and relay depth k. In both figures, asynchronous consensus with $f = 1$ is impossible for $k = 1$, but possible for $k = 2$.

1 Introduction

The fault-tolerant consensus problem proposed by Lamport et al. [20] has been studied extensively under different point-to-point network models, including complete networks (e.g., [20, 12]) and undirected networks (e.g., [13, 11]). Recently, many works are exploring various consensus problems in directed networks, e.g., [7, 5, 16], including our own work [23, 25, 22]. More precisely, these works address the problem in *incomplete directed* networks, i.e., not every pair of nodes is connected by a channel, and the channels are not necessarily bi-directional. We will often use the terms *graph* and *network* interchangeably. In this work, we explore the crash-tolerant approximate consensus problem in *asynchronous* incomplete networks under different restrictions on *topology knowledge* – where we assume that each node knows all its neighbors of at most k-hop distance – and *relay depth* – the maximum number of hops that information (or a message) can be propagated. These constraints are common in large-scale networks to avoid memory overload and network congestion, e.g., neighbor table and Time-to-live (TTL) (or hop limit) in the Internet Protocol (IP). We consider both undirected and directed graphs in this paper.

Motivation. Prior results [23] showed that exact crash-tolerant consensus is solvable in *synchronous* networks with only one-hop knowledge and relay depth 1, i.e., each node only needs to know its immediate incoming and outgoing neighbors, and no message needs to be relayed (or forwarded). Such a local algorithm is of interest in practice due to low deployment cost and low message complexity. In *asynchronous* undirected networks, there exists a simple flooding-based algorithm adapted from [13, 11] that achieves approximate consensus with up to f crash faults if the network satisfies $(f + 1)$ node-connectivity[2] and $n > 2f$, where n is the number of nodes. However, these two conditions are *not* sufficient for an *iterative* algorithm with one-hop knowledge and relay depth 1, in which each node maintains a state and exchanges state values with only one-hop neighbors in each iteration.

Consider Figure 1a, which is a ring network of four nodes. There is no iterative algorithm with one-hop knowledge and relay depth 1 under one crash fault. The adversary can divide the nodes into disjoint sets $\{a, b\}$ and $\{c, d\}$ such that the communication delay across sets is so large that a thinks d has crashed, and d thinks a has crashed, and similarly for the pair b and c. As a result, no exchange of state values is possible across the sets in the execution; hence, consensus is not possible (a more precise discussion in Section 3). On the other hand, suppose each node has two-hop knowledge, i.e., a complete topology knowledge in this network, and relay depth 2. Then a knows that it will be able to receive state values from at least two of the other nodes since the node connectivity is 2, and up to one node may fail.

[2] For brevity, we will simply use the term "connectivity" in the presentation below.

Following this observation, it is easy to design a flooding-based algorithm in the ring network based on [13, 11]. This example shows that both topology knowledge and relay depth affect the feasibility of asynchronous approximate consensus.

Interestingly, increasing connectivity alone does *not* make iterative algorithm feasible. In the full version [21], we show that no fault-tolerant approximate consensus algorithm with one-hop topology and relay depth 1 exists in the network in Figure 1b, which has two sparsely-connected cliques of size $n/2$ and connectivity $n/2 - 1$. Motivated by these observations, this work addresses the following question in *asynchronous* systems:

> What is a tight condition on the underlying communication graphs for achieving approximate consensus if each node has only a k-hop topology knowledge and relay depth k'?

The problem. We consider the asynchronous approximate consensus problem. The system consists of n nodes, of which at most f nodes may crash. Each node is given an input, and after a finite amount of time, each fault-free node should produce an output, which satisfies *validity* and *agreement* conditions (formally defined later). Intuitively, the state at fault-free nodes must be in the range of all the inputs, and are guaranteed to be within ϵ of each other for some $\epsilon > 0$ after a sufficiently large number of rounds.

In [23], we presented Condition CCA (Crash-Consensus-Asynchronous, see definition in Section 2) and showed that it is necessary and sufficient on the underlying directed graphs for achieving *approximate* consensus in asynchronous systems [23]. The approximate consensus algorithms in prior work [23, 13, 11] are based on flooding (i.e., relay depth n) and assume that each node has n-hop topology knowledge. However, such an algorithm is *not* practical in a large-scale network, since, (i) nodes' local memory may not be large enough to store the entire network, (ii) flooding-based algorithms (e.g., [23, 13, 11]) incur prohibitively high message overhead for each phase, and (iii) complete topology knowledge may require a high deployment and configuration cost. Therefore, we explore algorithms that only require "local" knowledge and limited message relay.

Contributions. We identify tight conditions on the graphs under different assumptions on topology knowledge and relay depth. Particularly, we have the following results:

- *Limited Topology Knowledge and Relay Depth* (Section 3): We consider the case with k-hop topology knowledge and relay depth k. The family of algorithms that captures these constrains are iterative k-hop algorithms – nodes only have topology knowledge of their k-hop neighborhoods, and propagate state values to nodes that are at most k-hops away. Note that *no* other information is relayed. For iterative k-hop algorithms, we derive a family of tight conditions, namely Condition k-CCA for $1 \le k \le n$, for solving *approximate* consensus in directed networks. To prove the tightness of the conditions, we propose a family of iterative algorithms called k-LocWA and show how the convergence time and the message complexity of those algorithms is affected by k, providing the respective upper bounds.
- *Topology Discovery and Unlimited Relay Depth* (Section 4): We consider the case with one-hop topology knowledge and relay depth n. In other words, nodes initially only know their immediate incoming and outgoing neighbors, but nodes can flood the network, learn (some part of) the topology, and eventually solve consensus based on the learned topology. We show that Condition CCA from [23] is also sufficient in this case. Since we assume only one-hop knowledge, our result implies that Condition CCA is tight for any k-hop topology knowledge. One contribution that may be of independent interest is

a *topology discovery* mechanism to learn and “estimate” the topology in asynchronous directed networks with crash faults. Such a discovery mechanism will be useful for self-stabilization and reconfiguration of a large-scale system.

In Section 5, we discuss fault-tolerance implications of the derived conditions and Condition CCA. We also discuss how to speed up our algorithms in terms of real time delay.

Related Work. There is a large body of work on fault-tolerant consensus. Here, we discuss related works exploring consensus in different assumptions on graphs. Fischer et al. [13] and Dolev [11] characterized necessary and sufficient conditions under which Byzantine consensus is solvable in *undirected* graphs. In synchronous systems, Charron-Bost et al. [7, 8] solved *approximate* crash-tolerant consensus in dynamic directed networks using local averaging algorithms, and in the asynchronous setting, Charron-Bost et al. [7, 8] addressed approximate consensus with crash faults in *complete* graphs which are necessarily undirected. We solve the problem in *incomplete directed* graphs in asynchronous systems. Moreover, in [7, 8], nodes are *constrained* to only have the one-hop topology knowledge. We study different types of algorithms, including the ones that allow nodes to learn the topology (i.e., we allow topology discovery).

There were also works studying limited topology knowledge. Su and Vaidya [22] identified the condition for solving synchronous Byzantine consensus using a variation of k-hop algorithms. Alchieri et al. [1] studied the synchronous Byzantine problem under *unknown* participants. We consider *asynchronous* systems in this work. Nesterenko and Tixeuil [17] studied the topology discovery problem in the presence of Byzantine faults in *undirected* networks, whereas we present a solution that works in *directed* networks with crash faults.

Extensive prior works studied graph properties for other similar problems in the presence of Byzantine failures, such as (i) Byzantine approximate consensus in directed graphs using “local averaging” algorithms wherein nodes only have one-hop neighborhood knowledge (e.g., [25, 24, 22, 26, 10]), (ii) Byzantine consensus with unknown participants [1], (iii) Byzantine consensus with *authentication* in *undirected* networks [3]. These papers only consider synchronous systems, and our algorithms and analysis are significantly different from those developed for Byzantine algorithms, and (iv) consensus problems in *synchronous* dynamic networks where the adversary can change the network topology. In this line of work, impossibility results for consensus and k-set agreement are given in [4, 6] and sufficiency is guaranteed by requiring a period of stability, during which certain nodes are strongly connected; the first tight condition for the feasibility of consensus and broadcast is presented in [9]. Additionally, in [2], Byzantine corruptions and a dynamic node set is assumed and a $O(\log^3 n)$-round randomized algorithm is presented. Our work is different from all these works because of the assumption of asynchronous systems and limited topology knowledge.

2 Preliminary

Before presenting the results, we introduce our system model, some terminology, and our prior results from [23] to facilitate the discussion.

System Model. The point-to-point message-passing network is *static*, and it is represented by a simple *directed* graph $G(\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the set of n nodes, and $\mathcal{E}$ is the set of directed edges between the nodes in $\mathcal{V}$. The communication links are reliable. We assume that $n \geq 2$, since the consensus problem for $n = 1$ is trivial. Node i can transmit messages to another node j directly if directed edge (i, j) is in $\mathcal{E}$. Each node can send messages to itself as well;

however, for convenience, we exclude self-loops from set $\mathcal{E}$. We will use the terms *edge* and *link* interchangeably.

Up to f nodes may suffer crash failures in an execution. A node that suffers a crash failure simply stops taking step (i.e., fail-stop model). We consider the *asynchronous* message-passing communication, in which a message may be delayed arbitrarily but eventually delivered if the receiver node is fault-free. We assume that the adversary has both the control of crashing nodes and delaying messages at any point of time during the execution.

Terminology. Upper case letters are used to name sets. Lower case italic letters are used to name nodes. All paths used in our discussion are directed paths.

Node j is said to be an incoming neighbor of node i if $(j,i) \in \mathcal{E}$. Let N_i^- be the set of incoming neighbors of node i, i.e., $N_i^- = \{j \mid (j,i) \in \mathcal{E}\}$. Define N_i^+ as the set of outgoing neighbors of node i, i.e., $N_i^+ = \{j \mid (i,j) \in \mathcal{E}\}$.

For set $B \subseteq \mathcal{V}$, node i is said to be an incoming neighbor of set B if $i \notin B$, and there exists $j \in B$ such that $(i,j) \in \mathcal{E}$. Given subsets of nodes A and B, set B is said to have k incoming neighbors in set A if A contains k distinct incoming neighbors of B.

▶ **Definition 1.** Given disjoint non-empty subsets of nodes A and B, $A \stackrel{x}{\Rightarrow} B$ if B has at least x distinct incoming neighbors in A. When it is not true that $A \stackrel{x}{\Rightarrow} B$, we will denote that fact by $A \stackrel{x}{\not\Rightarrow} B$.

Approximate Consensus. For the approximate consensus problem (e.g., [12, 15, 23]), it is usually assumed that each node i maintains and regularly updates a *state*, with $v_i[p]$ denoting the p-th update of the state of node i. In asynchronous systems, value $v_i[p]$ is also called the state of node i at the end of phase (or iteration) p. The initial state of node i, $v_i[0]$, is equal to the initial input provided to node i. At the start of phase p ($p > 0$), the state of node i is $v_i[p-1]$.

Let $U[p]$ and $\mu[p]$ be the maximum and the minimum state at nodes that have not crashed by the end of phase p. Then, a *correct* approximate consensus algorithm needs to satisfy the following two conditions:

- *Validity*: $\forall p > 0, U[p] \le U[0]$ and $\mu[p] \ge \mu[0]$; and
- *Convergence*: $\lim_{p\to\infty} U[p] - \mu[p] = 0$.

Equivalently the Convergence condition can be stated as:

$$\forall \epsilon > 0, \text{ there exists a phase } p_\epsilon \text{ such that for } p > p_\epsilon, U[p] - \mu[p] < \epsilon.$$

Towards facilitating the study of the number of phases needed for convergence and the corresponding message complexity, observe that convergence with respect to a specific ϵ must be considered. Therefore we will also use the following convergence notion.

- ϵ-Convergence: $\exists p_\epsilon$, $\forall p \ge p_\epsilon$, $U[p] - \mu[p] \le \epsilon$.

Remark on Termination. The variation of approximate consensus defined above, also appears in the literature under the name of *asymptotic consensus* (cf. [14]). The main difference is that in the original approximate consensus problem defined in [12], termination for any ϵ is also required. However, we stress that the algorithms we present can trivially be extended to achieve termination using an approach similar to the ones presented in [12, 15]. In these works and in most of the approximate consensus literature, the problem is solved by iterative algorithms of similar structure and each node can locally compute an upper bound on the number of iterations that are needed for ϵ-convergence. Thus termination is easily guaranteed.

Prior Result. In [23], we identified necessary and sufficient conditions on the underlying communication graphs $G(\mathcal{V}, \mathcal{E})$ for achieving *crash-tolerant consensus* in directed networks. The theorem below, presented in [23], states that condition **CCA** (Crash-Consensus-Asynchronous) is tight for approximate consensus under full topology knowledge and relay depth. Observe that, naturally, the impossibility result holds regardless of those parameters.

▶ **Theorem 2** (from [23]). *Approximate crash-tolerant consensus in asynchronous systems, under full topology knowledge and relay depth, is feasible iff for any partition L, C, R of $\mathcal{V}$, where L and R are both non-empty, either $L \cup C \overset{f+1}{\Rightarrow} R$ or $R \cup C \overset{f+1}{\Rightarrow} L$. **(Condition CCA)***

3 Limited Topology Knowledge and Relay Depth

In this section, we study how topology knowledge and the relay depth affect the *tight* conditions on the directed communication network. Particularly, we consider the case with k-hop topology knowledge and relay depth k for $1 \leq k \leq n$. Prior works (e.g., [23, 13, 11]) assumed that each node has n-hop topology knowledge and relay depth n. However, in large-scale networks, such an assumption may not be realistic. Partial knowledge models have been recently explored in [18, 19]. We are interested in algorithms that only require nodes to exchange a small amount of information within their local neighborhoods. One other benefit is that these algorithms do not require flooding [23] or all-to-all communication [13, 11] in each asynchronous phase.

We are interested in iterative k-hop algorithms – nodes only have topology knowledge in their k-hop neighborhoods, and propagate state values to nodes that are at most k-hops away. We introduce a family of conditions, namely Condition k-CCA for $1 \leq k \leq n$, which we prove necessary and sufficient for achieving asynchronous approximate consensus, through the use of iterative k-hop algorithms. The results presented in this section also imply how the parameter k affects the *tight* conditions on the directed networks. To the best of our knowledge, two prior papers [1, 22] examined a similar problem – *synchronous* Byzantine consensus. In [22], Su and Vaidya identified the condition under different relay depths. Alchieri et al. [1] studied the problem under *unknown* participants. The technique developed for asynchronous consensus in this section is significantly different.

Observe that since the system is asynchronous, any algorithm has to be event-oriented. For this reason, we define a locally verifiable condition *WAIT*, which dictates the end of the reception step and the start of the state update step as seen below. Different *WAIT* conditions need to be defined for each algorithm we consider.

Iterative *k*-hop Algorithms. The iterative algorithms considered here have relay depth k and require each node i to perform the following three steps in *asynchronous* phase t:

1. *Transmit*: Transmit messages of the form $(v_i[t-1], t-1)$ to nodes that are reachable from node i via *at most k hops away*, where $v_i[t-1]$ is the current state value, which is accompanied by the phase tag $t-1$. If node i is an intermediate node on the route of some message, then node i forwards that message as instructed by the source;
2. *Receive*: Until a condition *WAIT* is satisfied, receive messages from the nodes that can reach node i via at most k hops. Denote by $R_i[t]$ the set of messages that node i received at phase t; and
3. *Update*: Once condition *WAIT* is satisfied, update state using a transition function Z_i, where Z_i is a part of the specification of the algorithm, and takes as input the set $R_i[t]$. i.e.,

$$v_i[t] := Z_i(R_i[t], v_i[t-1]) \quad \text{at node } i$$

Note that (i) no exchange of topology information takes place in this class of algorithms, and (ii) each node's state only propagates within its k-hop neighborhood. For a node i, its *k-hop incoming neighbors* are defined as the nodes j which are connected to i by a directed path in G that has $\leq k$ hops. The notion of k-hop outgoing neighbors is defined similarly.

Technique. The algorithms presented in this section are motivated by prior work [12, 22] including our own work [23]. The algorithms are iterative and simple; thus, the proof structure shares some similarity with prior work [12, 23, 25].

Generally speaking, the proof proceeds as following: (i) nodes are divided into two disjoint sets, say L and R so that nodes have "closer" state values in each set; (ii) because each node receives an adequate set of messages, we show that under any delay and crash scenarios, at least one non-crashed node in either L or R will receive one message from the other set of nodes in each phase; and (iii) after enough phases, the value of all non-crashed nodes in either L or R will move "closer" to the values in the other set. Two key novelties in this paper are: identifying the "adequate set" of messages that needs to be received before updating local state in each asynchronous phase, and showing that even with limited k-hop propagation, some node is still able to receive messages from the other set (satisfying the above point (ii)).

3.1 $k = 1$ Case

To initiate the study, we first consider the one-hop case, where each node only knows its one-hop incoming and outgoing neighbors. The following notion is crucial for the characterization of graphs in which asynchronous approximate consensus is feasible with relay depth 1.

▶ **Definition 3** ($A \rightarrow B$)**.** Given disjoint non-empty subsets of nodes A and B, we will use the notation $A \rightarrow B$ if there exists a node i in B such that i has at least $f + 1$ distinct incoming neighbors in A. When it is not true that $A \rightarrow B$, we will denote that fact by $A \not\rightarrow B$.

Condition 1-CCA, presented below proves to be necessary and sufficient for achieving asynchronous approximate consensus with relay depth 1. Note that 1-CCA requires the existence of a single node that has at least $f + 1$ incoming neighbors, while CCA requires the distinct incoming neighbors of the corresponding set to be at least $f + 1$.

▶ **Definition 4** (Condition 1-CCA)**.** For any partition L, C, R of $\mathcal{V}$, where L and R are both non-empty, either $L \cup C \rightarrow R$ or $R \cup C \rightarrow L$.

The necessity of Condition 1-CCA for the specific class of iterative 1-hop algorithms, is similar to the necessity proof of Condition CCA in [23] and is presented in the full version [21].

For sufficiency, we present Algorithm LocWA (Local-Wait-Average), which is inspired by Algorithm WA [23] [3]. Note that LocWA utilizes only one-hop information. Recall that by definition, no message relay with depth greater than 1 is allowed. In Algorithm LocWA, $heard_i[p]$ is the set of one-hop incoming neighbors of i from which i has received values during phase p. Each node i performs the averaging operation to update its state value when Condition 1-WAIT below holds for the first time in phase p.

[3] The main difference lies in the WAIT condition and the fact that no relay of messages takes place in LocWA. Also, the termination of LocWA can be dealt with as argued in the corresponding remark in Section 2.

Algorithm 1: LocWA for node $i \in \mathcal{V}$.

$v_i[0] :=$ input at node i

For phase $p \geq 1$:

* On entering phase p:

$R_i[p] := \{v_i[p-1]\}$

$heard_i[p] := \{i\}$

Send message $(v_i[p-1], i, p)$ to all the outgoing neighbors

* When message (h, j, p) is received for the *first time*:

$R_i[p] := R_i[p] \cup \{h\}$ // $R_i[p]$ is a multiset

$heard_i[p] := heard_i[p] \cup \{j\}$

* When Condition *1-WAIT* holds for the first time in phase p:

$$v_i[p] := \frac{\sum_{v \in R_i[p]} v}{|R_i[p]|} \tag{1}$$

Enter phase $p+1$

Condition 1-WAIT. The condition is satisfied at node i, in phase p, when $|heard_i[p]| \geq |N_i^-| - f$, i.e., when i has not received values from a set of at most f incoming neighbors.

To prove the correctness of LocWA, we will use the supplementary definitions below.

▶ **Definition 5.** For disjoint sets A, B, $in(A \to B)$ denotes the set of all the nodes in B that each have at least $f+1$ incoming edges from nodes in A. When $A \not\to B$, define $in(A \to B) = \emptyset$. Formally, $in(A \to B) = \{\ v \mid v \in B \text{ and } \ f+1 \leq |N_v^- \cap A|\ \}$.

▶ **Definition 6.** For *non-empty disjoint* sets A and B, set A is said to *propagate to* set B in l steps, where $l > 0$, if there exist sequences of sets $A_0, A_1, A_2, \cdots, A_l$ and $B_0, B_1, B_2, \cdots, B_l$ (propagating sequences) such that

- $A_0 = A$, $B_0 = B$, $A_l = A \cup B$, $B_l = \emptyset$, $B_\tau \neq \emptyset$ for $\tau < l$, and
- for $0 \leq \tau \leq l-1$, (i) $A_\tau \to B_\tau$; (ii) $A_{\tau+1} = A_\tau \cup in(A_\tau \to B_\tau)$; and (iii) $B_{\tau+1} = B_\tau - in(A_\tau \to B_\tau)$.

Observe that A_τ and B_τ form a partition of $A \cup B$, and for $\tau < l$, $in(A_\tau \to B_\tau) \neq \emptyset$. We say that set A propagates to set B if there is a propagating sequence for some steps l as defined above. Note that the number of steps l in the above definition is upper bounded by $n - f - 1$, since set A must be of size at least $f+1$ for it to propagate to B; otherwise, $A \not\to B$.

Now, we present two key lemmas whose proofs are presented in the full version [21]. In the discussion below, we assume that G satisfies Condition 1-CCA.

▶ **Lemma 7.** *For any partition A, B of $\mathcal{V}$, where A, B are both non-empty, either A propagates to B, or B propagates to A.*

The lemma below states that the interval to which the states at all the fault-free nodes are confined shrinks after a finite number of phases of Algorithm LocWA. Recall that $U[p]$ and $\mu[p]$ denote the maximum and minimum states at the fault-free nodes at the end of the p-th phase and. We also denote with $F[p]$, the nodes that have not computed value $v[p]$ in phase p, i.e., nodes in $F[p]$ have crashed before computing $v[p]$.

▶ **Lemma 8.** *Suppose that at the end of the p-th phase of Algorithm LocWA, $\mathcal{V}$ can be partitioned into non-empty sets R and L such that (i) R propagates to L in l steps, and (ii)*

the states of fault-free nodes in $R - F[p]$ are confined to an interval of length $\leq \frac{U[p]-\mu[p]}{2}$. Then, with Algorithm LocWA,

$$U[p+l] - \mu[p+l] \;\leq\; \left(1 - \frac{\alpha^l}{2}\right)(U[p] - \mu[p]), \qquad \textit{where} \quad \alpha = \min_{i \in \mathcal{V}} \frac{1}{|N_i^-|} \tag{2}$$

Using lemma 8 and simple algebra, we can prove the following Theorem. For the sake of space, we present only a proof sketch. The complete proof is deferred to the full version [21].

▶ **Theorem 9.** *If $G(\mathcal{V}, \mathcal{E})$ satisfies Condition 1-CCA, then Algorithm LocWA achieves both Validity and Convergence.*

Proof Sketch. To prove the Convergence of LocWA, we show that given any $\epsilon > 0$, there exists τ such that $U[t] - \mu[t] \leq \epsilon, \forall t \geq \tau$. Consider p-th phase, for some $p \geq 0$. If $U[p] - \mu[p] = 0$, then the algorithm has already converged; thus, we consider only the case where $U[p] - \mu[p] > 0$. In this case, we can partition $\mathcal{V}$ into two subsets, A and B, such that, for each fault-free node $i \in A$, $v_i[p] \in \left[\mu[p], \frac{U[p]+\mu[p]}{2}\right)$, and for each fault-free node $j \in B$, $v_j[p] \in \left[\frac{U[p]+\mu[p]}{2}, U[p]\right]$. (Full proof in [21], identifies how to partition the nodes.) By Lemma 7, we have that either A propagates to set B or B propagates to A. In both cases above, we have found two non-empty sets $L = A$ (or $L = B$) and $R = B$ (or $L = A$) partitioning $\mathcal{V}$ and satisfy the hypothesis of Lemma 8, since R propagates to L and the states of all fault-free nodes in R are confined to an interval of length $\leq \frac{U[p]-\mu[p]}{2}$. The theorem is then proven by using simple algebra and the fact that the interval to which the states of all the fault-free nodes are confined shrinks after a finite number of phases. ◀

3.2 General k Case

Now, consider the case when each node only knows its k-hop neighbors and the relay depth is k. In the following, we generalize the notions presented above to the k-hop case. For node i, denote by $N_i^-(k)$ the set of i's k-hop incoming neighbors, For a set of nodes A, let N_A^- be the set of A's one-hop incoming neighbors. Formally, $N_A^- = \{i \mid i \in \mathcal{V} - A, \text{ and } \exists j \in A, (i,j) \in \mathcal{E}\}$. Next we define the relation $A \to B$ for the k-hop case.

▶ **Definition 10** ($A \to_k B$)**.** Given disjoint non-empty subsets of nodes A and B, we will say that $A \to_k B$ holds if there exists a node i in B for which there exist at least $f + 1$ node-disjoint paths of length at most k from distinct nodes in A to i. More formally, if $\mathcal{P}_i^A(k)$ is the family of all sets of k-length node-disjoint paths (with i being their only common node) initiating in A and ending in node i, $A \to_k B$ means that $\exists i \in B, \max_{P \in \mathcal{P}_i^A(k)} |P| \geq f + 1$.

▶ **Definition 11** (Condition k-CCA)**.** For any partition L, C, R of $\mathcal{V}$, where L and R are both non-empty, either $L \cup C \to_k R$ or $R \cup C \to_k L$.

The necessity of Condition k-CCA for achieving asynchronous approximate consensus through an iterative k-hop algorithm holds analogously with the one-hop case, where a set of x incoming neighbors of node i has to be replaced with a set of x distinct nodes that reach i through disjoint paths. For sufficiency, we next present a generalization of Algorithm LocWA for the k-hop case. There are two differences between Algorithms k-LocWA and LocWA: (i) nodes transmit their state to all their k-hop outgoing neighbors, and (ii) Algorithm k-LocWA relies on the generalized version of Condition 1-WAIT, presented below.

Algorithm 2: k-LocWA for node $i \in \mathcal{V}$.

$v_i[0] :=$ input at node i
For phase $p \geq 1$:
* On entering phase p:
    $d_i[p] := 1$
    $R_i[p] := \{v_i[p-1]\}$
    $heard_i[p] := \{i\}$
    Send message $(v_i[p-1], i, p)$ to nodes in $N_i^+(k)$, all k-hop outgoing neighbors[a]
* When message (h, j, p) is received for the *first time*:
    $R_i[p] := R_i[p] \cup \{h\}$ // $R_i[p]$ is a multiset
    $heard_i[p] := heard_i[p] \cup \{j\}$
* When Condition k-$WAIT$ holds for the first time in phase p:
    $v_i[p] := \frac{\sum_{v \in R_i[p]} v}{|R_i[p]|}$
    Enter phase $p+1$

[a] For brevity, we do not specify how the network routes the messages within the k-hop neighborhood – this can be achieved by using local flooding through tagging a hop counter in each message.

Condition k-WAIT. For $F_i \subseteq N_i^-(k)$, we denote with $reach_i^k(F_i)$ the set of nodes that have paths of length $l \leq k$ to node i in G_{V-F_i}. That is, the set of k-hop incoming neighbors of i that remain connected with i even when all nodes in set F_i crash. The condition is satisfied at node i, in phase p if there exists $F_i \subseteq N_i^-(k)$ with $|F_i[p]| \leq f$ such that $reach_i^k(F_i[p]) \subseteq heard_i[p]$.

Correctness of Algorithm k-LocWA. Proving the correctness of k-LocWA follows a similar reasoning of the correctness of LocWA. The key here is to identify Condition k-CCA and Condition k-WAIT so that the proof structure remains almost identical. To adapt the arguments to the general case, one should define the analogous $in(A \rightarrow_k B)$ definition based on the general $A \rightarrow_k B$ notion.

▶ **Definition 12.** For disjoint sets A, B, $in(A \rightarrow_k B)$ denotes the set of all the nodes i in B such that there exist at least $f+1$ incoming disjoint paths of length at most k from distinct nodes in $N_i^- \cap A$ to i. When $A \not\rightarrow_k B$, define $in(A \rightarrow_k B) = \emptyset$. Formally, in the terminology of Definition 10: $in(A \rightarrow B) = \{i \in B : \max\{|p| : p \in P_i^A(k)\} \geq f+1\}$

The following proof sketch outlines necessary adaptations for general k case proof.

▶ **Theorem 13.** *Approximate crash-tolerant consensus in an asynchronous system using iterative k-hop algorithms is feasible iff G satisfies Condition k-CCA.*

Proof Sketch. Having defined the basic notion $in(A \rightarrow_k B)$, Definition 6 of the notion A *propagates to* B is the same for the k-hop case. Intuitively, if A *propagates to* B, information will be propagated gradually from A to B in l steps. Any faulty set of f nodes will *not* be able to block propagation from A to a specific node $i \in B$ because the definition of $in(A \rightarrow_k B)$ guarantees that i will receive information from at least $f+1$ disjoint paths if it has not crashed. A difference with the $k=1$ case is that for every of the l steps needed to propagate from A to B, k communication steps will be required in the worst case, since information may be propagated through paths of length k. Lemma 8 is intuitively the same since it is based on the general propagation notion but value α which is defined based on

the number of incoming neighbors will now be defined on the number of k-hop incoming neighbors, i.e., $\alpha_k = \min_{i \in \mathcal{V}} \frac{1}{|N_i^-(k)|}$. The main correctness proof remains essentially the same since it repeatedly makes use of the abstract propagation notion between various sets, without focusing on how the values are propagated. ◀

3.3 Condition Relation and Convergence Time Comparison

Next, we first compare the feasibility of approximate consensus for different values of k by presenting a relation among the various k-CCA conditions as well as their relation with Condition CCA from [23].

Condition Relation

Intuitively, achieving approximate consensus for a lower k requires the existence of more paths in the graph; this can be observed by definition and is summarized in the following theorem.

▶ **Theorem 14.** *For values $k, k' \in \mathbb{N}$ with $k \leq k'$, Condition k-CCA implies Condition k'-CCA.*

Proof. Let Condition k-CCA hold and assume, without loss of generality that $L \cup C \rightarrow_k R$ holds for a partition L, C, R. This means that there exists a node i in R that has at least $f+1$ incoming disjoint paths of length at most k initiating from distinct nodes in $L \cup C$. Consequently, the same $f+1$ paths will consist of i's incoming disjoint paths of length at most k', since $k' \geq k$, and thus, $L \cup C \rightarrow_{k'} R$ which means that k'-CCA holds. ◀

We next show that Condition CCA is equivalent to Condition n-CCA. The proof illustrates how the locally defined Condition k-CCA naturally coincides with the globally defined condition CCA in the extreme case.

▶ **Theorem 15.** *Condition CCA is equivalent to Condition n-CCA.*

Proof. It is easy to see that Condition n-CCA implies Condition CCA. If Condition CCA is violated in G, then Condition n-CCA does not hold either, since L and R have at most f one-hop incoming neighbors.

Now, we show the other direction. Assume for the sake of contradiction that Condition CCA holds but Condition n-CCA does not. Then, there exists a partition L, C, R with $L, R \neq \emptyset$ such that $L \cup C \not\rightarrow_k R$ and $R \cup C \not\rightarrow_k L$. Since Condition CCA holds, we have that either $L \cup C \overset{f+1}{\Rightarrow} R$ or $R \cup C \overset{f+1}{\Rightarrow} L$. Now consider the case that $L \cup C \overset{f+1}{\Rightarrow} R$ and $R \cup C \overset{f+1}{\not\Rightarrow} L$. This means that $|N_R^-| \geq f+1$ and $|N_L^-| \leq f$. The case of $L \cup C \overset{f+1}{\not\Rightarrow} R$ and $R \cup C \overset{f+1}{\Rightarrow} L$ is symmetrical and the case of $L \cup C \overset{f+1}{\Rightarrow} R$ and $R \cup C \overset{f+1}{\Rightarrow} L$ can be proved by applying the argument below once for set R and once for set L.

Let i be the node in R with the maximum number m of disjoint paths initiating from distinct nodes in $V - R$ (as implied by Definition 10). The fact $L \cup C \not\rightarrow_k R$ implies that $m \leq f$. Subsequently, $|N_R^-| \geq f+1$ implies that the set $A = N_R^- - N_i^-(n)$ is non-empty (the maximal subset of N_R^- which does not contain any n-hop incoming neighbors of i). Let $B = N_A^+(n) \cap R$ be the set of all the outgoing n-hop neighbors of all nodes $j \in A$ confined in the set R. By definition of B and A, it holds that $N_i^-(n) \cap B = \emptyset$. We can now create a new partition $L' = L, C' = C \cup B, R' = R - B$ by moving B from R to C. For partition L', C', R' it holds that $L', R' \neq \emptyset$ since $i \in R'$ and $L' = L$. Moreover, it holds that (i) $|N_{R'}^-| \leq f$, since

$|N_{R'}^-| = |N_R^- - A|$ and $A \neq \emptyset$; and (ii) $|N_L^-| \leq f$ since $L = L'$. The latter points imply that $R \cup C \overset{f+1}{\not\Rightarrow} L$ and $L \cup C \overset{f+1}{\not\Rightarrow} R$, which yield a contradiction to the hypothesis that Condition CCA holds. This completes the proof. ◀

Convergence Time Comparison

We derive upper bounds on the number of *asynchronous* phases needed for ϵ-convergence of Algorithm k-LocWA and its message complexity up to this ϵ-convergence point p_ϵ. These upper bounds are functions of values ϵ, k, f, n and $\delta = U[0] - \mu[0]$ which are naturally expected to affect the convergence time and message complexity. Moreover, since the bounds depend on k, it provides a way to compare the convergence time and message complexity of Algorithms k-LocWA for different values of k. Next, we present the upper bound on the convergence time of k-LocWA, the proof of the theorem is deferred to the full version [21].

▶ **Theorem 16** (Convergence-time complexity)**.** *The number of phases required by Algorithm k-LocWA to ϵ-converge is* $O\left(\dfrac{(n-f)\log \epsilon/\delta}{\log\left(1 - \frac{\alpha_k^{n-f-1}}{2}\right)}\right)$.

Comparison of Algorithms k-LocWA Convergence. Observe that the above bound decreases, as the maximum number of k-hop incoming neighbors increases, since $\alpha_k = \min_{i \in \mathcal{V}} \dfrac{1}{|N_i^-(k)|}$. Since the maximum number of k-hop incoming neighbors increases with k we have that for $k' > k$, Algorithm k'-LocWA ϵ-converges faster than k-LocWA by a factor implied by the bound. Moreover, given the upper bound on phases for ϵ-convergence of Theorem 16 we can easily derive an upper bound on the message complexity of k-LocWA as is shown in [21].

4 Topology Discovery and Unlimited Relay Depth

In this section, we consider the case with one-hop topology knowledge and relay depth n. In other words, nodes initially only know their immediate incoming and outgoing neighbors, but nodes can flood the network and learn the topology. The study of this case is motivated by the observation that full topology knowledge at each node (e.g., [23, 13, 11]) requires a much higher deployment and configuration cost. We show that Condition CCA from [23] is necessary and sufficient for solving approximate consensus with one-hop neighborhood knowledge and relay depth n in asynchronous directed networks. Compared to the iterative k-hop algorithms in Section 3, the algorithms in this section are *not* restricted in the sense that nodes can propagate any messages to all the reachable nodes.

The necessity of Condition CCA is implied by our prior work [23]. The algorithms presented below are again inspired by Algorithm WA from [23]. The main contribution is to show how each node can learn "enough" topology information to solve approximate consensus – this technique may be of interest in other contexts as well. In the discussion below, we present an algorithm that works in any directed graph that satisfies Condition CCA.

Algorithm LWA. The idea of Algorithm LWA (Learn-Wait-Average) is to piggyback the information of incoming neighbors when propagating state values. Then, each node i will locally construct an *estimated* graph $G^i[p]$ in every phase p, and check whether Condition

Algorithm 3: LWA for node $i \in \mathcal{V}$.

$v_i[0] :=$ input at node i
$G^i[0] := G_{N_i^- \Rightarrow i}$
For phase $p \geq 1$:
* On entering phase p:
  $R_i[p] := \{v_i[p-1]\}$
  $heard_i[p] := \{i\}$
  Send message $(v_i[p-1], N_i^-, i, p)$ to all the outgoing neighbors
* When message (h, N, j, p) is received for the *first time*:
  $R_i[p] := R_i[p] \cup \{h\}$ // $R_i[p]$ is a multiset
  $heard_i[p] := heard_i[p] \cup \{j\}$
  $G^i[p] := G^i[p] \cup G_{N \Rightarrow j}$ [a]
  Send message (h, N, j, p) to all the outgoing neighbors
* When Condition n-*WAIT* holds on $G^i[p]$ for the first time in phase p:
  $v_i[p] := \frac{\sum_{v \in R_i[p]} v}{|R_i[p]|}$
  $G^i[p+1] := G_{N_i^- \Rightarrow i}$ // "Reset" the learned graph
  Enter phase $p+1$

[a] $G_1(\mathcal{V}_1, \mathcal{E}_1) \cup G_2(\mathcal{V}_2, \mathcal{E}_2) \equiv G_3(\mathcal{V}_3, \mathcal{E}_3)$, where $\mathcal{V}_3 = \mathcal{V}_1 \cup \mathcal{V}_2$ and $\mathcal{E}_3 = \mathcal{E}_1 \cup \mathcal{E}_2$. Note that this is *not* a multiset, there is only one copy of each node or edge.

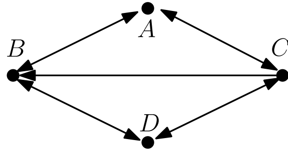
n-WAIT holds in $G^i[p]$ or not. Note that $G^i[p]$ may not equal to G, as node i may not receive messages from some other nodes due to asynchrony or failures. We say Condition n-WAIT holds in the local estimated graph $G^i[p](\mathcal{V}^i[p], \mathcal{E}^i[p])$ if **there exists** a set $F_i[p] \subseteq \mathcal{V}^i[p] - \{i\}$, where $|F_i[p]| \leq f$, such that $reach_i'(F_i[p]) \subseteq heard_i[p]$. Here, $reach_i'(F_i)$ is the set of nodes that have paths to node i in the subgraph induced by the nodes in $\mathcal{V}^i[p] - F_i[p]$ for $F_i[p] \subseteq \mathcal{V}^i[p] - \{i\}$ and $|F_i[p]| \leq f$.

Recall that N_i^- denotes the set of i's one-hop incoming neighbors. Given a set of nodes N and node i, we also use the notation $G_{N \Rightarrow i}$ to describe a directed graph consisting of nodes $N \cup \{i\}$ and set of directed edges from each node in N to i. Formally, $G_{N \Rightarrow i} = (N \cup \{i\}, E')$, where $E' = \{(j, i) \mid j \in N\}$.

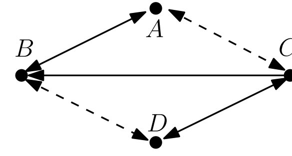
Correctness of Algorithm LWA. The key lemma to prove the correctness of Algorithm WA in [23] is to show that for any pair of nodes that have not crashed in phase p, they must receive a state value from at least one common node. In the full version [21], we show that Algorithm LWA achieves the same property. Intuitively, if Condition n-WAIT does not hold in the local estimated graph $G^i[p]$, then node i knows it can learn more states in phase p. Also, when Condition n-WAIT is satisfied in $G^i[p]$, there exists a scenario that node i cannot receive any more information; hence, it should not wait for any more message. This is why the Algorithm LWA allows each node to learn enough state values to achieve approximate consensus. We rely on this observation to prove the correctness in [21]. Algorithm LWA works on undirected graphs as well, as is shown in the full version [21].

5 Discussion

In asynchronous systems, the real time communication delay is arbitrary but finite. In a formal framework, it is common to assume that execution proceeds in rounds representing real time intervals, but the nodes do not have knowledge of the round index. To model the



(a) Graph G, i-CCA holds for any $i \in \{1, \ldots, 4\}$ and $f = 1$.



(b) Arbitrary delay in directed edges $(A, C), (C, A), (B, D), (D, B)$

Figure 2 Real time delay example.

worst-case real time delay in the execution of a system we can use the notion of *delay scenario* which is a description of the delays, incurring on the communication through all edges of the network. The delivery delay of a message sent over a channel e will be described by the number of rounds (amount of real time) that are needed for the delivery to be completed.

We first compare the real time performance of Algorithms k-LocWA for different values of k with respect to the real time delay. Specifically we show that there is a case where Algorithm LocWA terminates each phase in one round (one interval of real time), while it may take arbitrary number of rounds for Algorithm 2-LocWA to terminate phase 1.

▶ **Example 17.** Consider the graph of Figure 2a. For $f = 1$, it is easy to verify that Condition 1-CCA holds, which implies that Conditions i-CCA, for $i \in \{1, \ldots, n\}$ hold. Assume that the delivery of messages through *directed* edges $(A, C), (C, A), (B, D), (D, B)$ is delayed by d rounds while the communication in all the other edges is instant (one round). For ease of presentation assume that no node crashes. Then, in an execution of Algorithm LocWA, it is clear that every node i will finish phase t in time t. On the other hand, in an execution of Algorithm 2-LocWA, node D will only receive a message from C in one round, since (C, B) is a directed edge, and delay on edges (A, C) and (B, D) is d. in this case, D will not be able to decide before round d, the first round where Condition 2-WAIT will be satisfied. Specifically, for the first phase it will hold that $reach^2_D \subseteq heard_D[1]$ only after round d since, if D considers $F_D = \{B\}$ as a possible corruption set, it has to wait for a message from A which will be propagated by C and setting $F_D = \{C\}$, it has to wait for a message from B. For similar reasons, the same holds for nodes A, C. Since d may be an arbitrary integer, there is a delay scenario where the ϵ-convergence time for Algorithm 2-LocWA is arbitrarily larger than the ϵ-convergence time of Algorithm LocWA.

Strong version of k-LocWA with respect to real time. As shown in the full version [21], the k-WAIT condition of k-LocWA algorithm can be strengthened such that, for $k' \geq k$ and any ϵ, Algorithm k'-LocWA will ϵ-converge faster than Algorithm k-LocWA. This can be achieved by condition strong k-WAIT: wait until $\bigvee_{i=1}^{k} (i\text{-WAIT}) = true$ for the first time.

References

1 EduardoA.P. Alchieri, AlyssonNeves Bessani, Joni Silva Fraga, and Fabíola Greve. Byzantine Consensus with Unknown Participants. In *Principles of Distributed Systems*, volume 5401 of *Lecture Notes in Computer Science*, pages 22–40. Springer Berlin Heidelberg, 2008. doi:10.1007/978-3-540-92221-6_4.

2 John Augustine, Gopal Pandurangan, and Peter Robinson. Fast Byzantine Agreement in Dynamic Networks. In *Proceedings of the 2013 ACM Symposium on Principles of*

Distributed Computing, PODC '13, pages 74–83, New York, NY, USA, 2013. ACM. doi:10.1145/2484239.2484275.

3 Piyush Bansal, Prasant Gopal, Anuj Gupta, Kannan Srinathan, and Pranav Kumar Vasishta. Byzantine agreement using partial authentication. In *Proceedings of the 25th international conference on Distributed computing*, DISC'11, pages 389–403, Berlin, Heidelberg, 2011. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=2075029.2075079.

4 Martin Biely, Peter Robinson, and Ulrich Schmid. Agreement in Directed Dynamic Networks. In Guy Even and Magnús M. Halldórsson, editors, *Structural Information and Communication Complexity*, pages 73–84, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

5 Martin Biely, Peter Robinson, Ulrich Schmid, Manfred Schwarz, and Kyrill Winkler. Gracefully Degrading Consensus and k-Set Agreement in Directed Dynamic Networks. In Ahmed Bouajjani and Hugues Fauconnier, editors, *Networked Systems*, pages 109–124, Cham, 2015. Springer International Publishing.

6 Martin Biely, Peter Robinson, Ulrich Schmid, Manfred Schwarz, and Kyrill Winkler. Gracefully degrading consensus and k-set agreement in directed dynamic networks. *Theoretical Computer Science*, 726:41–77, 2018. doi:10.1016/j.tcs.2018.02.019.

7 Bernadette Charron-Bost, Matthias Függer, and Thomas Nowak. Approximate Consensus in Highly Dynamic Networks. *CoRR*, abs/1408.0620, 2014. URL: http://arxiv.org/abs/1408.0620, arXiv:1408.0620.

8 Bernadette Charron-Bost, Matthias Függer, and Thomas Nowak. Approximate Consensus in Highly Dynamic Networks: The Role of Averaging Algorithms. In *Proceedings, Part II, of the 42Nd International Colloquium on Automata, Languages, and Programming - Volume 9135*, ICALP 2015, pages 528–539, New York, NY, USA, 2015. Springer-Verlag New York, Inc. doi:10.1007/978-3-662-47666-6_42.

9 Étienne Coulouma and Emmanuel Godard. A Characterization of Dynamic Networks Where Consensus Is Solvable. In Thomas Moscibroda and Adele A. Rescigno, editors, *Structural Information and Communication Complexity*, pages 24–35, Cham, 2013. Springer International Publishing.

10 S. M. Dibaji, H. Ishii, and R. Tempo. Resilient Randomized Quantized Consensus. *IEEE Transactions on Automatic Control*, PP(99):1–1, 2017. doi:10.1109/TAC.2017.2771363.

11 Danny Dolev. The Byzantine Generals Strike Again. *Journal of Algorithms*, 3(1), March 1982.

12 Danny Dolev, Nancy A. Lynch, Shlomit S. Pinter, Eugene W. Stark, and William E. Weihl. Reaching approximate agreement in the presence of faults. *J. ACM*, 33:499–516, May 1986. doi:10.1145/5925.5931.

13 Michael J. Fischer, Nancy A. Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. In *Proceedings of the fourth annual ACM symposium on Principles of distributed computing*, PODC '85, pages 59–70, New York, NY, USA, 1985. ACM. doi:10.1145/323596.323602.

14 Heath LeBlanc, Haotian Zhang, Xenofon D. Koutsoukos, and Shreyas Sundaram. Resilient Asymptotic Consensus in Robust Networks. *IEEE Journal on Selected Areas in Communications*, 31(4), 2013. doi:10.1109/JSAC.2013.130413.

15 Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

16 Alexandre Maurer, Sébastien Tixeuil, and Xavier Défago. Reliable Communication in a Dynamic Network in the Presence of Byzantine Faults. *CoRR*, abs/1402.0121, 2014. URL: http://arxiv.org/abs/1402.0121, arXiv:1402.0121.

17 Mikhail Nesterenko and Sébastien Tixeuil. Discovering Network Topology in the Presence of Byzantine Faults. In *Structural Information and Communication Complexity*, pages 212–226, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

18 Aris Pagourtzis, Giorgos Panagiotakos, and Dimitris Sakavalas. Reliable broadcast with respect to topology knowledge. *Distributed Computing*, 30(2):87–102, 2017. doi:10.1007/s00446-016-0279-6.

19 Aris Pagourtzis, Giorgos Panagiotakos, and Dimitris Sakavalas. Reliable Communication via Semilattice Properties of Partial Knowledge. In Ralf Klasing and Marc Zeitoun, editors, *Fundamentals of Computation Theory - 21st International Symposium, FCT 2017, Bordeaux, France, September 11-13, 2017, Proceedings*, volume 10472 of *Lecture Notes in Computer Science*, pages 367–380. Springer, 2017. doi:10.1007/978-3-662-55751-8_29.

20 M. Pease, R. Shostak, and L. Lamport. Reaching Agreement in the Presence of Faults. *J. ACM*, 27(2):228–234, April 1980. doi:10.1145/322186.322188.

21 Dimitris Sakavalas, Lewis Tseng, and Nitin H. Vaidya. Asynchronous Crash-Tolerant Approximate Consensus in Directed Graphs: Topology Knowledge. *CoRR*, abs/1803.04513, 2018. arXiv:1803.04513.

22 Lili Su and Nitin Vaidya. Reaching Approximate Byzantine Consensus with Multi-hop Communication. In Andrzej Pelc and Alexander A. Schwarzmann, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 9212 of *Lecture Notes in Computer Science*, pages 21–35. Springer International Publishing, 2015. doi:10.1007/978-3-319-21741-3_2.

23 Lewis Tseng and Nitin H. Vaidya. Fault-Tolerant Consensus in Directed Graphs. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 451–460, New York, NY, USA, 2015. ACM. doi:10.1145/2767386.2767399.

24 Lewis Tseng and Nitin H. Vaidya. Iterative approximate Byzantine consensus under a generalized fault model. In *In International Conference on Distributed Computing and Networking (ICDCN)*, January 2013.

25 Nitin H. Vaidya, Lewis Tseng, and Guanfeng Liang. Iterative Approximate Byzantine Consensus in Arbitrary Directed Graphs. In *Proceedings of the thirty-first annual ACM symposium on Principles of distributed computing*, PODC '12. ACM, 2012.

26 H. Zhang and S. Sundaram. Robustness of distributed algorithms to locally bounded adversaries. In *Proceedings of ACC 2012, the 31st American Control Conference*, 2012.

Hybrid Fault-Tolerant Consensus in Asynchronous and Wireless Embedded Systems

Wenbo Xu
Technische Universität Braunschweig, Braunschweig, Germany
wxu@ibr.cs.tu-bs.de

Signe Rüsch
Technische Universität Braunschweig, Braunschweig, Germany
ruesch@ibr.cs.tu-bs.de

Bijun Li
Technische Universität Braunschweig, Braunschweig, Germany
bli@ibr.cs.tu-bs.de

Rüdiger Kapitza[1]
Technische Universität Braunschweig, Braunschweig, Germany
kapitza@ibr.cs.tu-bs.de

Abstract

Byzantine fault-tolerant (BFT) consensus in an asynchronous system can only tolerate up to $\lfloor \frac{n-1}{3} \rfloor$ faulty processes in a group of n processes. This is quite a strict limit in certain application scenarios, for example a group consisting of only 3 processes. In order to break through this limit, we can leverage a hybrid fault model, in which a subset of the system is enhanced and cannot be arbitrarily faulty except for crashing. Based on this model, we propose a randomized binary consensus algorithm that executes in complete asynchrony, rather than in partial synchrony required by deterministic algorithms. It can tolerate up to $\lfloor \frac{n-1}{2} \rfloor$ Byzantine faulty processes as long as the trusted subsystem in each process is not compromised, and terminates with a probability of one. The algorithm is resilient against a strong adversary, i. e. the adversary is able to inspect the state of the whole system, manipulate the delay of every message and process, and then adjust its faulty behaviour during execution.

From a practical point of view, the algorithm is lightweight and has little dependency on lower level protocols or communication primitives. We evaluate the algorithm and the results show that it performs promisingly in a testbed consisting of up to 10 embedded devices connected via an ad hoc wireless network.

2012 ACM Subject Classification Software and its engineering → Software fault tolerance

Keywords and phrases Distributed system, consensus, fault tolerance

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.15

1 Introduction

Fault-tolerant consensus is one of the fundamental problems in distributed systems. It is an essential component to build more sophisticated distributed applications. Especially with the rapid growth of wireless embedded devices in recent years, systems tend to work cooperatively to achieve a common goal [16, 11, 14]. This requires novel consensus algorithms to adapt to

[1] This work is part of the DFG Research Unit Controlling Concurrent Change, funding no. FOR 1800 and grant no. KA 3171/5-1.

© Wenbo Xu, Signe Rüsch, Bijun Li, and Rüdiger Kapitza;
licensed under Creative Commons License CC-BY
22nd International Conference on Principles of Distributed Systems (OPODIS 2018).
Editors: Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira; Article No. 15; pp. 15:1–15:16

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

the corresponding system features and application requirements that are inherently different from the quite well-explored wired settings. Firstly, network connectivity is fragile and an upper bound of communication delay may not be guaranteed. Thus an asynchronous system model must be considered. Secondly, the system could be running in an open environment in contrast to more or less static and well maintained environments such as data centers. Furthermore processes might be exposed to extreme temperature, radiation, physical damage and malicious attacks. All these factors lead to a higher demand for fault-resilience. In short, the system has to tolerate as many faulty processes as possible – and not only crashes, but also Byzantine faults. Thirdly, because of the limited resources, the consensus algorithm should have low complexity and less assumptions about low-level network protocols.

In this work we present a novel binary consensus algorithm, Trusted Ben-Or, which is tailored for embedded wireless systems. It is fault-tolerant and can work in a completely asynchronous system. Due to the fact that no deterministic consensus algorithm exists in an asynchronous system with faulty processes [13], Trusted Ben-Or uses randomization to overcome this impossibility. In the end it ensures that all correct processes can terminate with the probability of 1.

Moreover, in an asynchronous system a total of n processes can only tolerate up to $\lfloor \frac{n-1}{3} \rfloor$ Byzantine processes in order to achieve consensus [8]. This is quite a strict constraint, and it is impossible for a group of only 3 processes to reach BFT consensus. So another important goal of our work is to increase the number of tolerable faulty processes utilizing a slightly extended system model. Many modern processors, including the ones used in the context of embedded systems [3], provide nowadays trusted execution environments, which can be utilized to build a trusted subsystem protected by hardware. The trusted subsystem is tamper-proof and cannot be compromised by a Byzantine host, except for crashing. This technique can be leveraged to prevent equivocation, i. e. sending contradictory messages to different recipients. As a result, the tolerable processes can be increased to $\lfloor \frac{n-1}{2} \rfloor$. The idea that a subsystem is more trustworthy than the remainder of the system can be categorized as a hybrid fault model [21].

In summary, the contributions of this paper include:

- We propose an asynchronous hybrid fault-tolerant consensus algorithm, Trusted Ben-Or, which can tolerate up to $\lfloor \frac{n-1}{2} \rfloor$ faulty processes among n processes.
- The algorithm is resilient against a strong adversary model, which means that the adversary can inspect the state of every process and message, and can arbitrarily reorder the message delivery of every process. We provide a correctness proof, which is the first complete proof under this system model to our knowledge.
- Trusted Ben-Or is tailored for wireless embedded systems for its simplicity and low complexity. Every message is sent via broadcast to make full use of the transmission medium. The communication does not require encryption, nor complex communication primitives such as reliable broadcast, nor TCP-like protocols that could be unavailable in certain application domains. Because of the trusted subsystem, the message authentication can use symmetric encryption, which is more efficient than asymmetric digital signatures.
- We implement Trusted Ben-Or and evaluate it in a real wireless ad hoc environment instead of in a pure simulation. The results are promising and comparable to Turquois [19], another well-known wireless ad hoc BFT consensus algorithm.
- We discuss some common issues regarding the termination of randomized BFT consensus algorithms, and point out that some algorithms might not be able to terminate in a strong adversary model.

The rest of the paper is organized as follows: Section 2 defines the system model and the problem statement. Section 3 explains the Trusted Ben-Or algorithm in details. The

correctness proof is given in Section 4. Section 5 discusses the optimization against omission failures and common issues of randomized BFT consensus algorithms. Section 6 shows the evaluation results. Section 7 discusses the related work, and Section 8 concludes the paper.

2 System Model and Problem Definition

In this section we give the system model and the correctness criteria of the consensus problem.

2.1 Processes and Asynchronous System

The system consists of n processes numbering from 1 to n. It is an asynchronous system in which the delay of message transmission or process execution is unbounded. We firstly consider a reliable communication so that every message transmitted between correct processes (defined below) is eventually delivered. Later we will discuss the message omission issue.

For the purpose of presentation, we assume a *global wall-clock*, which is nevertheless not available to the processes. We also model the algorithm execution as a discrete event simulation and assume a virtual *scheduler*. At each time (clock tick) t, the scheduler chooses a process to take a step, e. g. to deliver a message, or to execute a line of the algorithm, etc. We call each step an *event*.

2.2 Strong Adversary Model and Trusted Subsystem

There exists an adversary working against the system. At any time t, the adversary can compromise the current scheduled process to behave abnormally. Such faulty behaviours include: skip some execution steps or stop working permanently (crash), update state or send messages not according to the algorithm, send messages to only a subset of recipients, etc. A process is marked as Byzantine at the time when he performs the faulty behaviours, and will never be regarded as correct afterwards. All the processes not marked as Byzantine are called *correct* at time t. In the remaining of this paper, the statements like "a correct process does something" indicate that the process is still correct at the time when he does the thing. The adversary can compromise at most $f \leq \lfloor \frac{n-1}{2} \rfloor$ processes in total. For simplicity we assume $f = \lfloor \frac{n-1}{2} \rfloor$ in the rest of the paper.

The adversary is a *strong adversary*, meaning that he knows all the previous events, and is able to inspect the current state of every process and the content of every message in the network at any time. He also has the full control over the scheduler to schedule the events arbitrarily. In this way, the adversary can adapt his behaviour to the current system state, leading to a maximum attack power.

However, each process is equipped with a trusted subsystem that cannot be compromised by the adversary and will always operate honestly, unless it is crashed. Furthermore, the adversary does not know the secret key(s) stored in the subsystem, neither can he break the cryptographic mechanisms. As a result, the adversary cannot bypass the trusted subsystem to forge any messages authenticated by the subsystem. As we will show later, the trusted subsystem is used to implement a hybrid fault model [21] similar to *identical Byzantine* [4].

▶ Remark. A strong adversary seems impractical in real-life scenarios. However, for a distributed algorithm, it is impossible to predict the execution order and message delivery order because of race conditions. There could be corner cases that will hinder the algorithm. In fact, a strong adversary model indicates that each of these corner cases is already tackled. Otherwise, people need to explain why these cases cannot or can hardly happen.

2.3 Problem Definition

Every process can propose a binary value 0 or 1. (A faulty process can propose both values, or no value at all. Any abnormal value other than 0 and 1 will be ignored by correct processes.) The randomized consensus algorithm is called correct, if it fulfils the following properties:

- *Agreement:* if two correct processes decide, they decide the same value.
- *Validity:* if a correct process decides v, $v \in \{0, 1\}$ and is proposed by at least $\lfloor \frac{n}{4} \rfloor + 1$ processes.
- *Termination:* any process decides with the probability of 1, or he becomes Byzantine.

Note that the validity implies that the decision can be proposed merely by Byzantine processes. This is different from the *strong validity* in most $3f + 1$ consensus problems, which requires that if all correct processes propose v, they must also decide v. However, strong validity is impossible in an asynchronous system with $n < 3f + 1$, so the requirement must be relaxed in our case. Yet it is adequate to certain applications, in which both values are acceptable and the processes only need to agree on a common one. Or some extra mechanisms can be applied to detect and eliminate the faulty value (e. g. [10]). Since these can be application-dependent and do not belong to the consensus, we omit the detailed discussion here.

3 Trusted Ben-Or Algorithm

Trusted Ben-Or is a randomized consensus algorithm consisting of several asynchronous rounds and each round has two phases, as shown in Algorithm 1. In the first phase, every process broadcasts a *P-message* to propose its current value, together with a flag indicating whether this value is taken deterministically (D-GET) or from a coin flip (R-GET). Upon receiving $\lceil \frac{n+1}{2} \rceil$ P-messages (including his own), a process enters the second phase by broadcasting a *V-message* to vote. He votes for $v \in \{0, 1\}$ if all received P-messages have the same value v, otherwise he votes for a default value $\perp$. The process then waits for $\lceil \frac{n+1}{2} \rceil$ V-messages. If all of them vote for the same value $v \in \{0, 1\}$, the process can decide on v. If at least one V-message votes for $v \in \{0, 1\}$ while the others vote for $\perp$, the process updates his own value to v and sets the flag to D-GET. Otherwise, i. e. all received V-messages vote for $\perp$, the process flips a coin to get a random number, and sets the flag to R-GET.

There is an initialization round (line 4-9), in which each process broadcasts his initial proposal and picks the majority from the received quorum to start round 1.

3.1 Message Authentication and Trusted Coin in the Subsystem

One critically harmful Byzantine fault is equivocation, i. e. sending inconsistent messages to different recipients in a broadcast. Equivocation can be prevented by reliable broadcast [7], which takes several communication rounds and requires $n > 3f$. It can also be avoided by simply using a strict monotonic counter in the trusted subsystem of every process. Every message must be authenticated together with a counter value, and each value can only be used once. More technical details can be referred to works such as [17].

The algorithm also requires a random bit (line 15), so an unbiased trusted coin is placed inside the trusted subsystem. This prevents a Byzantine process from arbitrarily manipulating the result of the coin. Moreover, the message authentication and random number generation should be integrated as an atomic operation. Otherwise, a Byzantine process can repeatedly toss the trusted coin until it obtains the desired result.

Algorithm 1 TRUSTED BEN-OR algorithm.

```
1  v_D ← v_p                                              /* Store last D-get value */
2  φ ← 0                                                  /* phase number */
3  flag ← D-GET
4  broadcast ⟨INIT, φ, p, v_D⟩
5  wait for ⌈(n+1)/2⌉ valid ⟨INIT, φ, i, v_i, flag_i⟩ messages from different processes
6  if among them exist ≥ ⌈(n+1)/2⌉/2 messages with value 0
7      v_D ← 0
8  else
9      v_D ← 1
10 φ ← 1
11 loop forever:
12     if flag = D-GET                                    /* P-phase */
13         broadcast ⟨PR, φ, p, v_D, D-GET⟩ with certificate (see Section 3.2)
14     else
15         broadcast ⟨PR, φ, p, coin(), R-GET⟩ with certificate
16     wait for ⌈(n+1)/2⌉ valid ⟨PR, φ, i, v_i, flag_i⟩ messages from different processes
17     if all messages of carry the same v:               /* V-phase */
18         broadcast ⟨VO, φ, i, v⟩ with certificate
19     else:
20         broadcast ⟨VO, φ, i, ⊥⟩ with certificate
21     wait for ⌈(n+1)/2⌉ valid ⟨VO, φ, i, ∗⟩ messages from different processes
22     if all messages of above line have the same value v ≠ ⊥:
23         decide v
24     if received at least one ⟨VO, φ, i, v_i⟩ with v_i ≠ ⊥:    /* Update */
25         v_D ← v_i
26         flag ← D-GET
27     else:
28         flag ← R-GET
29     φ ← φ + 1
```

The trusted subsystem maintains a unique identifier uid, a monotonically increasing counter value u, and secrete key(s) to calculate message authentication codes. The keys cannot be disclosed to the non-trusted part of the system. It provides the following APIs:

- `authenticate`(m, u): Takes a message m and a counter value u; Requires u greater than its last accepted value; Outputs an authentication code based on $m||uid||u$.
- `authenticate_with_coin`(m, ptr, u): Takes an extra pointer pointing to a bit of m; Requires u greater than its last accepted value; Firstly fills the bit where ptr points to with the trusted coin toss result, then outputs an authentication code based on $m||uid||u$.
- `verify`(m, uid, u, AC): Checks whether the authentication code AC is generated based on $m||uid||u$ by the trusted subsystem uid.

Now we define the corresponding counter value u that is uniquely bound to every message. More specifically, $INIT$ message must be authenticated with counter value $u = 0$. Every P-message of round ϕ must have $u = [\phi|0]$, where "|" is the separation of the least significant bit and higher bits. And every V-message must have $u = [\phi|1]$. Note that at line 15 of Algorithm 1, $\langle PR, \phi, p, \texttt{coin}(), \textit{R-GET}\rangle$ is authenticated by invoking `authenticate_with_coin`, and the result of the a coin toss is filled into the place where `coin`() stands.

3.2 Message Certificate and Validation

Every message needs to be proved that it is congruent with the algorithm specification. To achieve this, a process is required to provide a set of previously received messages, named

certificate, when he broadcasts a new message. The certificate is piggybacked with the newly sent message. A message is called *valid* only if it includes the correct certificate. The certificate of every message is defined as follows:

1. $\langle INIT, 1, *, v, D\text{-}GET\rangle$ is valid if $v \in \{0,1\}$ without any certificate.
2. $\langle PR, 1, *, v, D\text{-}GET\rangle$ requires $(\lfloor \frac{n}{4} \rfloor + 1)\langle INIT, 0, *, v\rangle$ for $v = 0$, or $(\lfloor \frac{n+2}{4} \rfloor + 1)$ $\langle INIT, 0, *, v\rangle$ for $v = 1$.[2]
3. $\langle PR, \phi, *, v, D\text{-}GET\rangle(\phi > 1)$ requires $\lceil \frac{n+1}{2} \rceil \langle PR, \phi - 1, *, v, *\rangle$.
4. $\langle PR, \phi, *, v, R\text{-}GET\rangle(\phi > 1)$ requires $\lceil \frac{n+1}{2} \rceil \langle VO, \phi - 1, *, \perp\rangle$.
5. $\langle VO, \phi, *, v\rangle$ with $v \in \{0,1\}$ requires $\lceil \frac{n+1}{2} \rceil \langle PR, \phi, *, v, *\rangle$. Furthermore, if there is one $\langle PR, \phi, *, v, D\text{-}GET\rangle$ in the certificate, the certificate of this message, i. e. $\lceil \frac{n+1}{2} \rceil \langle PR, \phi - 1, *, v, *\rangle$, must also be included.
6. $\langle VO, 1, *, \perp\rangle$ requires $\lceil \frac{n+1}{2} \rceil \langle PR, 1, *, *, D\text{-}GET\rangle$ with both 0 and 1 are proposed.
7. $\langle VO, \phi, *, \perp\rangle(\phi > 1)$ requires $\lceil \frac{n+1}{2} \rceil \langle PR, \phi, *, *\rangle$ with both 0 and 1 are proposed, plus $\lceil \frac{n+1}{2} \rceil \langle VO, \phi - 1, *, \perp\rangle$.

The certificates are carefully designed to ensure the correctness of the algorithm. There are two important properties. Firstly, a process can immediately validate a message based on the certificate, and does not rely on previously received messages. Secondly, the messages in the certificate do not require further certificate for themselves. Otherwise the message size will grow infinitely. However, this gives a Byzantine process the chance to bypass the validity check, because he can include some other invalid messages into the certificate, turning a faulty message into valid. But this issue does not affect the correctness, as we will see later.

The cases 1, 2, 4 and 6 are trivial as they directly follow the algorithm execution. The other three cases deserve some explanations.

- In case 3, if a correct process i sends a $\langle PR, \phi, i, v, D\text{-}GET\rangle(\phi > 1)$, he must have received at least one valid $\langle VO, \phi - 1, *, v\rangle$ (line 24). It is pointless to use this single message as a certificate, because this message itself can be invalid. But if i is correct, he must have validated the $\langle VO, \phi - 1, *, v\rangle$, which contains $\lceil \frac{n+1}{2} \rceil \langle PR, \phi - 1, *, v, *\rangle$ according to case 5. So he must include these $\lceil \frac{n+1}{2} \rceil$ messages into his own certificate.
- In case 5, $\langle VO, \phi, i, v\rangle$ implies that the process i has received $\lceil \frac{n+1}{2} \rceil \langle PR, \phi, *, v, *\rangle$ (line 17), which must be included in the certificate. The extra requirement is only necessary for the termination as we will prove later. If i is correct, he must have checked the validity of any $\langle PR, \phi, *, v, D\text{-}GET\rangle$ before putting it into certificate. So he is required to strip the certificate of that message and put into his own certificate.
- In case 7, the process sends $\langle VO, \phi, i, \perp\rangle(\phi > 1)$ because he has received $\lceil \frac{n+1}{2} \rceil \langle PR, \phi, *, *\rangle$ proposing different values (line 19). As we will see later, it is impossible to see both valid $\langle PR, \phi, *, 0, D\text{-}GET\rangle$ and $\langle PR, \phi, *, 1, D\text{-}GET\rangle$ in the same round $\phi > 1$, so one of the different values must come from a $\langle PR, \phi, *, *, R\text{-}GET\rangle$ caused by a coin flip, whose certificate is $\lceil \frac{n+1}{2} \rceil \langle VO, \phi - 1, *, \perp\rangle$. So i must strip the certificate from this *R-GET* message and include it into his own certificate.

According to the explanation above, we can conclude the following lemma:

▶ **Lemma 1.** *If a correct process is about to broadcast a message, he is able to attach a corresponding certificate to that message immediately.*

---

[2] Because $\lfloor \frac{n}{4} \rfloor + 1 = \lceil \frac{n+1}{2} \rceil / 2$, i. e. a majority of the quorum. And if a tie between 0 and 1 appears in the quorum, we choose 0 here. This can be modified to 1, depending on the application.

4 Correctness Proof

In this section we prove the correctness of Trusted Ben-Or.

4.1 Agreement

We first show that if any correct process decides any value v, then from the next round, no one can generate a valid P-message to propose another value.

▶ **Lemma 2.** *If a correct process decides v in round ϕ, the only valid P-message of $\phi + k$ is $\langle PR, \phi + k, *, v, D\text{-}GET\rangle$ for any $k > 0$.*

Proof. We start with $k = 1$. A correct process only decides v if there are $\lceil\frac{n+1}{2}\rceil\langle VO, \phi, *, v\rangle$. This excludes the existence of any valid $\langle PR, \phi+1, *, *, R\text{-}GET\rangle$ that requires $\lceil\frac{n+1}{2}\rceil\langle VO, \phi, *, \perp\rangle$ as certificate. Because 2 quorums or $\lceil\frac{n+1}{2}\rceil$ processes intersect with at least one process, the equivocation mechanism ensures that a process cannot vote for v and $\perp$ in the same round. And each $\langle VO, \phi, *, v\rangle$ contains $\lceil\frac{n+1}{2}\rceil\langle PR, \phi, *, v, *\rangle$ in its certificate. So for the same reason, $\langle PR, \phi+1, *, 1-v, D\text{-}GET\rangle$, which requires $\lceil\frac{n+1}{2}\rceil\langle PR, \phi, *, 1-v, *\rangle$, cannot become valid, either. On the other hand, $\langle PR, \phi + 1, *, v, D\text{-}GET\rangle$ can be valid because its certificate exists, making it to be the only valid P-message of $\phi + 1$.

Now assume that the only valid P-message of $\phi + k$ is $\langle PR, \phi + k, *, v, D\text{-}GET\rangle$ for some $k > 0$. Then all correct processes only broadcast $\langle PR, \phi + k, *, v, D\text{-}GET\rangle$. This makes $\langle PR, \phi+k+1, *, 1-v, D\text{-}GET\rangle$ invalid. Since all correct processes do not accept any invalid $\langle PR, \phi+k, *, 1-v, *\rangle$, they can only broadcast $\langle VO, \phi+k, *, v\rangle$. So $\langle PR, \phi+k+1, *, *, R\text{-}GET\rangle$ cannot be valid because less than $\lceil\frac{n+1}{2}\rceil$ vote for $\perp$. Thus $\langle PR, \phi + k + 1, *, v, D\text{-}GET\rangle$ is the only valid P-message. Using induction we can confirm this lemma. ◀

The agreement property directly ensues:

▶ **Theorem 3.** *No two correct processes decide differently.*

Proof. We prove it by contradiction. Suppose that two correct processes decide v in ϕ and $1-v$ in ϕ' respectively. Apparently $\phi \neq \phi'$, because $\lceil\frac{n+1}{2}\rceil\langle VO, \phi, *, v\rangle$ and $\lceil\frac{n+1}{2}\rceil\langle VO, \phi, *, 1-v\rangle$ cannot exist at the same time. Assume $\phi < \phi'$, according to Lemma 2, the only valid P-message of ϕ' is $\langle PR, \phi', *, v, D\text{-}GET\rangle$. But correct process decides $1-v$ in ϕ' means that he has received $\lceil\frac{n+1}{2}\rceil$ valid $\langle VO, \phi', i, 1-v\rangle$ messages certified with $\lceil\frac{n+1}{2}\rceil\langle PR, \phi', *, 1-v, *\rangle$. This leads to a contradiction, because no correct process will propose $1-v$ in a P-message. ◀

4.2 Termination

The proof of termination is similar to the proof of [2], but is more complex due to the Byzantine behaviours. We first show that every correct process is able to start any round, then prove that there is some "lucky" round in which every correct process can decide.

▶ **Lemma 4.** *Every correct process is able to start any round $\phi \geq 1$.*

Proof. It clearly applies for $\phi = 0$ and 1. Now assume that every correct process starts a round $\phi \geq 1$. According to Lemma 1, each correct process is able to assemble a certificate and broadcast a valid P-message. So eventually there are at least $\lceil\frac{n+1}{2}\rceil$ valid P-messages in the system, enabling correct processes to terminate the wait of line 16 and then broadcast a valid V-message. Again there are at least $\lceil\frac{n+1}{2}\rceil$ valid V-messages eventually, so every correct process can terminate the wait of line 21 and start the next round $\phi + 1$. Using induction we confirm that every correct process can start any round $\phi \geq 1$. ◀

▶ **Corollary 5.** *In every round ϕ, at least one of the three P-message forms is valid: $\langle PR, \phi, *, 0, D\text{-}GET\rangle$, $\langle PR, \phi, *, 1, D\text{-}GET\rangle$, $\langle PR, \phi, *, *, R\text{-}GET\rangle$. And at least one of the three V-message forms is valid: $\langle VO, \phi, *, 0\rangle$, $\langle VO, \phi, *, 1\rangle$, $\langle VO, \phi, *, \perp\rangle$*

In order to show a lucky round will eventually happen, we adopt the similar definition of [2], but with some modifications due to the presence of Byzantine faults:

▶ **Definition 6.** A value $v \in \{0, 1\}$ is ***ϕ*-major** at time t_0, if $\geq \lceil\frac{n+1}{2}\rceil$ processes have created the message $\langle PR, \phi, *, v, *\rangle$ and authenticated with the monotonic counter at t_0. A value $v \in \{0, 1\}$ is ***ϕ*-locked** at time t_0, if 1) no valid $\langle PR, \phi, *, 1 - v, *\rangle$ with a certificate exists before t_0 and 2) from t_0 on, no $\langle PR, \phi, *, 1 - v, *\rangle$ can be created, or such a message can never collect a certificate.

In other words, v is ϕ-locked at t_0 means that we are sure that no valid $\langle PR, \phi, *, 1 - v, *\rangle$ can exist at any time. Obviously, if v is ϕ-major or ϕ-locked at t_0, then v is also ϕ-major or ϕ-locked at any time $t \geq t_0$.

▶ **Lemma 7.** *If a value v is ϕ-locked at some time, then every correct process can decide v by the end of round $\phi + 1$.*

Proof. All correct processes will start round ϕ and they only propose v. They will not accept any P-message with value $1 - v$, since it is invalid. So they only vote for v, leading to less than $\lceil\frac{n+1}{2}\rceil\langle VO, \phi, *, \perp\rangle$, therefore $\langle VO, \phi + 1, *, \perp\rangle$ can never become valid. Neither $\langle VO, \phi + 1, *, 1 - v\rangle$ can be valid, because of the lack of certificate. According to Lemma 4 and Corollary 5, a correct process can complete collecting valid V-messages at line 21, and the only valid form is $\langle VO, \phi + 1, *, v\rangle$. As a result, every correct process can decide v by the end of $\phi + 1$. ◀

In the rest part we will show if the trusted random number generators of every process happen to generate a sequence of lucky results, one value will become locked. We start with the following lemma:

▶ **Lemma 8.** *If v is ϕ-major at t_0, and if the trusted random number generator happens to output v for every process creating $\langle PR, \phi + 1, *, v, R\text{-}GET\rangle$ at any time (can be before t_0), then v is $(\phi + 1)$-locked at t_0.*

Proof. There is no $\langle PR, \phi + 1, *, 1 - v, R\text{-}GET\rangle$ because of the trusted random number generator. And a $\langle PR, \phi+1, *, 1-v, D\text{-}GET\rangle$ cannot be valid any more, because v is already ϕ-major and no certificate containing $\lceil\frac{n+1}{2}\rceil\langle PR, \phi, *, 1 - v, *\rangle$ can exist. ◀

Starting from $\phi = 2$, we group every 3 rounds into an *epoch*. So the r-th epoch ($r \geq 1$) consists of rounds $3r - 1$, $3r$ and $3r + 1$. And we define two oracle functions. The oracles know the state of the whole system, but are not available to the processes. The `first_toss`(r, t) oracle returns the time $t_a \leq t$, when the first correct process executes line 15 to flip a coin to create $\langle PR, 3r, *, v, R\text{-}GET\rangle$ in round $3r$. If no correct processes ever did that, the oracle returns NaN. Note that the correct process i needs only to be correct until t_a. Also note that the function can return $t_a = t$, i. e. a correct process is executing line 15 exactly at time t.

The `lucky_coin`$(r, \phi, t) \to \{0, 1\}$ oracle assesses whether a random bit obtained in round ϕ, at time t is lucky or not. The return value of `lucky_coin`(r, ϕ, t) is defined as following:

- **(i)** `lucky_coin`$(r, 3r + 1, t)$ returns 1 for any t;
- **(ii)** `lucky_coin`$(r, 3r - 1, t)$ and `lucky_coin`$(r, 3r, t)$ return 1, if `first_toss`(r, t) returns t_a, and 0 is not $(3r - 1)$-major at time t_a;
- **(iii)** `lucky_coin`$(r, 3r - 1, t)$ and `lucky_coin`$(r, 3r, t)$ return 0 in cases other than (ii).

Obviously, as soon as a process (correct or Byzantine) has executed line 15 to flip the coin, we can immediately know whether the result is lucky or not. The reason is that the return values of both `first_toss`(r, t) and `lucky_coin`(r, ϕ, t) are determined merely by the events happened before or at t, and are independent of any future events.

We say an epoch is *lucky*, if every coin toss at time t of line 15 gets the consistent result of `lucky_coin`(r, ϕ, t). Let t_b be the time when every correct process has completed round $3r+1$. Such a t_b must exist (Lemma 4), and the system state has only three possibilities:

1. `first_toss`(r, t_b) returns NaN, meaning that no correct process ever tossed a coin;
2. `first_toss`(r, t_b) returns t_a and 0 is $(3r-1)$-major at time t_a;
3. `first_toss`(r, t_b) returns t_a and 0 is not $(3r-1)$-major at time t_a.

For the three cases, there are the following lemmas:

▶ **Lemma 9.** *(Case 1) If* `first_toss`(r, t_b) *returns* NaN*, then some value* v *is* $(3r+1)$*-locked at* t_b*.*

Proof. All correct processes have completed phase $3r$ and no one created $\langle PR, 3r, *, *, R\text{-}GET\rangle$, so they all must have created $\langle PR, 3r, *, v, D\text{-}GET\rangle$ with the same v.

If $v = 0$: because `lucky_coin`$(r, 3r, t)$ must return 0 due to its definition (case (iii)), so all $\langle PR, 3r, *, v', R\text{-}GET\rangle$ messages (must be created by Byzantine processes) have $v' = 0$. And there are no valid $\langle PR, 3r, *, 1, D\text{-}GET\rangle$ messages, so 0 is $3r$-locked, thus also $(3r+1)$-locked.

If $v = 1$: since all correct processes propose 1 in round $3r$, 1 is $3r$-major at time t_b. According to case (i) of the definition of `lucky_coin`, in round $3r+1$, the lucky coin always returns 1. So 1 is $(3r+1)$-locked because of Lemma 8. ◀

▶ **Lemma 10.** *(Case 2) If* `first_toss`(r, t_b) *returns* t_a *and 0 is* $(3r-1)$*-major at time* t_a*, then 0 is* $3r$*-locked at* t_b*.*

Proof. Because 0 is $(3r-1)$-major at time t_a, it is also $(3r-1)$-major at time t_b. For any time $t < t_a$, `lucky_coin`$(r, 3r, t)$ must return 0; for any time $t \geq t_a$, `lucky_coin`$(r, 3r, t)$ must return 0 as well. So every $\langle PR, 3r, *, v, R\text{-}GET\rangle$, whenever it is created, must have $v = 0$. According to Lemma 8, 0 must be $3r$-locked. ◀

▶ **Lemma 11.** *(Case 3) If* `first_toss`(r, t_b) *returns* t_a *and 0 is not* $(3r-1)$*-major at time* t_a*, then 1 is* $(3r+1)$*-locked at* t_b*.*

Proof. A correct process creates a $\langle PR, 3r, *, v, R\text{-}GET\rangle$ at time t_a (here $v = 1$). This indicates that he must have received $\lceil \frac{n+1}{2} \rceil \langle VO, 3r-1, *, \perp\rangle$, among which at least one is from a correct process. That process must have received at least one valid $\langle PR, 3r-1, *, 1, *\rangle$. This P-message cannot have the flag $R\text{-}GET$, because any coin tossed before t_a must get 0 (case (iii) of `lucky_coin`$(r, 3r-1, t)$). So there is one valid $\langle PR, 3r-1, *, 1, D\text{-}GET\rangle$, but there is never valid $\langle PR, 3r-1, *, 0, D\text{-}GET\rangle$. From time t_a on, the lucky coin only returns 1 for phase $3r-1$ according to case (ii), so there is no more $\langle PR, 3r-1, *, 0, R\text{-}GET\rangle$ after t_a. Thus the total number of $\langle PR, 3r-1, *, 0, R\text{-}GET\rangle$ is less than $\lceil \frac{n+1}{2} \rceil$ forever. So we can confirm that there is never a valid $\langle VO, 3r-1, *, 0\rangle$, because 1) it cannot include an invalid $\langle PR, 3r-1, *, 0, D\text{-}GET\rangle$ into the certificate (recall the definition of the certificate), and 2) the number of $\langle PR, 3r-1, *, 0, R\text{-}GET\rangle$ can never reach $\lceil \frac{n+1}{2} \rceil$.

Now that there are no valid $\langle VO, 3r-1, *, 0\rangle$, and the lucky coin only returns 1 for phase $3r$ after t_a, so no correct process will create $\langle PR, 3r, *, 0, *\rangle$. Eventually 1 will be $3r$-major. And because the lucky coin always returns 1 in round $3r+1$, 1 must be $(3r+1)$-locked because of Lemma 8. ◀

▶ **Lemma 12.** *If epoch r is lucky, all correct processes can decide no later than round $3r+2$.*

Proof. According to Lemmas 9, 10 and 11, some value v must be $3r$-locked or $(3r+1)$-locked at some time. So all correct processes can decide in round $3r+1$ or $3r+2$ (Lemma 7). ◀

▶ **Theorem 13.** *The probability that all correct processes decide is 1.*

Proof. An epoch is lucky if and only if all results from the random number generator coincide with the definition of `lucky_coin`. Each of these coincidences has the probability of 0.5, and is independent with each other. Every epoch may contain at most $3n$ random numbers, so the probability that an epoch is lucky is at least $(0.5)^{3n}$. So the probability that a lucky epoch eventually occurs is $1-(1-p^{3n})^{\infty}=1$. Lemma 12 ensures that all correct processes must decide immediately after such a lucky epoch. ◀

4.3 Validity

▶ **Theorem 14.** *If a correct process decides v, $v \in \{0,1\}$ and is proposed by at least $\lfloor \frac{n}{4} \rfloor + 1$ processes.*

Proof. All messages are required to carry a value $v \in \{0,1\}$, so correct processes cannot decide other values. Assume v is proposed by less than $\lfloor \frac{n}{4} \rfloor + 1$ processes. Then there is no valid $\langle PR, 1, *, v, * \rangle$ in the first round because of the lack of $INIT$ messages, which means v is 1-locked. Every correct process will decide $1-v$ by the end of round 2 (Lemma 7), so no correct one will decide v. ◀

5 Optimization and Discussion

Now we discuss some common issues in the proof of randomized consensus algorithms, as well as the optimization against omission failures, given that the reliable communication without message omissions is sometimes impractical in certain applications.

5.1 Randomization and Strong Adversary

Like most randomized and round-based algorithms, the termination of Trusted Ben-Or relies on a set of processes to luckily obtain the preferred coin values in certain rounds. The definition of a lucky coin value is not trivial, but we argue that this is necessary in a strong adversary model. As mentioned, the luckiness of a coin only depends on the current system state, but not on any future events. Otherwise, suppose that a coin is only lucky if something in the future happens, the adversary could take actions to prevent this from occurring. If an algorithm design violates this principle, it can hardly withstand a strong adversary.

We have considered several other algorithms (e. g. [10, 19]). Similar to Trusted Ben-Or, by the end of each round every process gets an updated value, either deterministically or randomly, and any two correct processes can only deterministically get the same value (0 or 1) in this round. So they conclude that, if all correct processes deterministically or randomly get the same v, they can decide. However, there is a corner case: if the correct processes start with different values, a strong adversary can hinder the termination indefinitely. Because by manipulating the message delivery order, the adversary can firstly let a correct node randomly obtain a value v, then lets another correct node deterministically update to $1-v$ in the same round. As a result, all the correct processes again have different values at the beginning of a new round.

5.2 Handling Omission Failures

In real-world networks, especially in wireless ad hoc networks, links are not always reliable and messages could get lost. In this case, we can consider a *fair-loss link* model, meaning that if a process p sends a message infinitely many times to q, q will then deliver the message infinitely many times. Trusted Ben-Or cannot directly work under a fair-loss link model, so we modify the algorithm by introducing another two tasks running in parallel to Algorithm 1. Task 1 is to periodically broadcast the last sent message. Task 2 is to let a process "jump" to a future round or phase, if it has received a valid message from that round or phase. More specifically, if a process receives a valid $\langle PR, \phi', *, v, \textit{D-GET}\rangle$ or $\langle VO, \phi', *, v\rangle$ that is more advanced than its current state, it will send a message with the same content, and update its state correspondingly. If a $\langle PR, \phi', *, *, \textit{R-GET}\rangle$ is received, besides updating the state, the process has to toss its own coin to create the $\langle PR, \phi, i, \texttt{coin}(), \textit{R-GET}\rangle$.

With this modification, the agreement of the algorithm still holds, because the correctness of Lemma 2 and Theorem 3 only relies on the equivocation prevention and certificate mechanism. However, the termination becomes problematic. Assume the corner case where a correct process receives all P-messages but misses all V-messages in each round. Then it can never decide. But we can guarantee that no correct process gets blocked at any round:

▶ **Lemma 15.** *At any time t_1, for any correct process in round ϕ_1, there is a time $t_2 > t_1$ when the process is in a new round $\phi_2 > \phi_1$, or it becomes Byzantine.*

Proof. Suppose there is a correct process p that stays forever in round ϕ_1 after time t_1. Then apparently there is no correct process entering any round $\phi_2 > \phi_1$ while also staying correct forever, otherwise it will periodically broadcast messages of round ϕ_2 or later rounds. Eventually p can receive at least one of them and can jump to a new round. So all other correct processes keep staying in rounds $\leq \phi_1$ after time t_1. Because p is infinitely broadcasting its message of ϕ_1, all correct processes will eventually receive it and will enter and stay in ϕ_1. Eventually at least one correct process can receive the messages from all correct processes, and can then enter $\phi_1 + 1$. This leads to a contradiction. ◀

Therefore, we have to assume a more relaxed model than fair-loss link, for instance every message gets an independent probability of omission (similar to [20]). The probability can vary from time to time but cannot always be 0. Then if there is a lucky epoch and all messages in this epoch as well as the following round are delivered, all the correct processes can decide. (They still have to handle the outdated V-messages for decision.) This assumption is close to the real network conditions. As we will see later, during our experiments we have observed regular packet losses (24%), but the algorithm can still terminate with relatively low latency.

6 Evaluation

Trusted Ben-Or is evaluated on a testbed consisting of 10 nodes of Raspberry Pi 3 (model B). Each node has a Quad-Core ARM Cortex-A53 1.2 GHz CPU, 1 GB RAM and a 2.4 GHz 802.11n wireless module. The trusted subsystem is built on top of the Open Portable Trusted Execution Environment (OP-TEE) [18] based on ARM TrustZone [3].[3] The algorithm is implemented in C++, except that the trusted functions (counter authentication and random number generation) are implemented in C because of lacking of C++ support in OP-TEE. For comparison, we implement Turquois on the same system as well.

[3] According to this disclaimer (`https://github.com/OP-TEE/build/blob/master/docs/rpi3.md`), Raspberry Pi does not provide an enough trust level. It is only used for demonstration purpose.

The 10 nodes are distributed in different rooms and the farthest distance is about 20 meters. They are connected with a wireless ad hoc network, and all messages of Trusted Ben-Or are sent via UDP multicast. The minimal, median and maximum of the round trip time of an ICMP ping message is 5.6 ms, 12.5 ms and 1356.7 ms respectively. With the *iperf3* tool we also test the UDP link, and the result between two nodes reports the jitter as 139.9 ms, which stands for a high variance of the communication delay, and 24% packet loss rate. So compared to a simulated network, this testbed can reflect a real network environment more closely. Because of the packet loss, we implemented the optimization of Section 5.2.

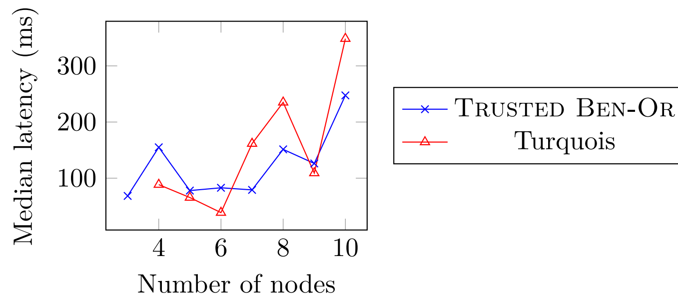
For the experiment, all nodes are connected to a signal machine via Ethernet, and wait for the start signal to start the consensus algorithm almost simultaneously. The nodes are equally divided to be assigned with 0 and 1 as their initial proposals. The performance is evaluated as the delay from the time when nodes start the algorithm until the non-faulty ones decide. For each system setting, we repeat the experiment 100 times.

Firstly the fault-free case is evaluated, and the results are presented in Figure 1. The curve of Turquois starts with 4 because it cannot work with 3 nodes. Figure 1a shows that the median of latency of Trusted Ben-Or has no significant difference compared to Turquois, especially when we take the high jitter value into account. With an increasing of group size, both algorithms tend to take longer time to terminate. The highest median latency of Trusted Ben-Or is 247.4 ms observed in a group with 10 nodes. Besides the median value, we also evaluate the variance of the latency as presented in Figure 1b. It shows that from $n = 8$, the variance increases notably in Trusted Ben-Or. Occasionally (with $\leq 10\%$ probability) it takes 2-3 seconds to terminate. In contrast, the variance of Turquois increases only slowly with n.

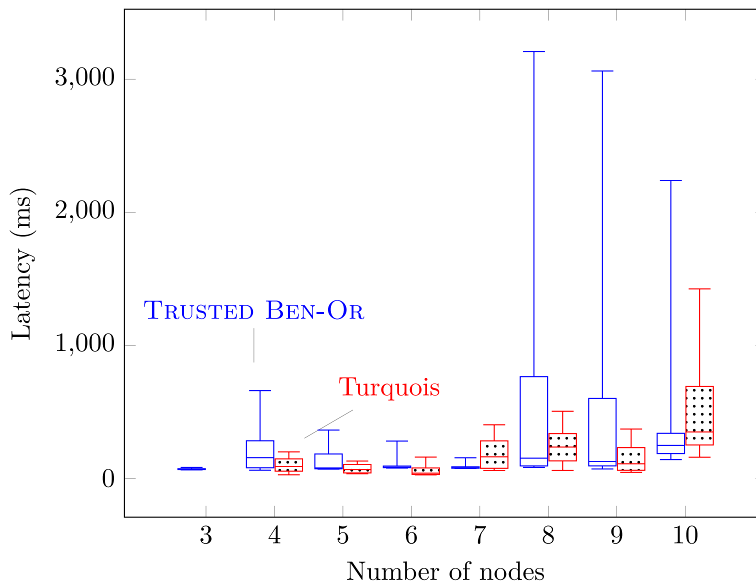
In the second experiment, we inject Byzantine faults into the system to evaluate the fault resilience of Trusted Ben-Or. More specifically, we let $\lfloor \frac{n-1}{2} \rfloor$ nodes act as Byzantine processes. Whenever they are about to send a value of 0 or 1 (if not from the trusted coin), they flip the value to the opposite and then send it; and if they are sending a $\perp$, they do not change it. Furthermore, we let Byzantine nodes not perform the validation at all, so an invalid message could still be included in the certificate of a message voting for $\perp$. As a result, there are fewer valid messages with value 0 or 1, while more valid messages voting for $\perp$. This will hinder the correct nodes achieving consensus. We inject faults into Turquois as well, but only $\lfloor \frac{n-1}{3} \rfloor$ nodes are Byzantine. Since there is no equivocation prevention mechanism in Turquois, we let faulty nodes always send two opposite values if their values are 0 or 1.

We compare the two algorithms with the existence of Byzantine faults and the results are shown in Figure 2. The difference between the median values of latency again is not significant, but both algorithms take longer to terminate. For example with $n = 10$, they need twice as much time as in the fault-free case. Regarding the variance, Turquois is not much affected by the Byzantine faults. But in Trusted Ben-Or the variance becomes higher. In the worst case, the latency could exceed 5 s with $n \geq 8$. This is partly because there are more Byzantine processes in Trusted Ben-Or.

We can conclude that for the most cases, Trusted Ben-Or does not introduce extra overhead when tolerating more faulty processes compared to Turquois. But the variance of Trusted Ben-Or is higher, and more sensitive to the Byzantine faults. Besides the different number of Byzantine processes, another reason is that Turquois has one more phase of message exchange in each round. At a first glance, this is counter-intuitive since one more phase means more communication delay. But in Turquois, in the second phase every process broadcasts its proposal and picks the majority as its vote in the third phase. So if a node is faster in progress, it is more likely to disseminate his proposal to the others because they



(a) Median of latency.



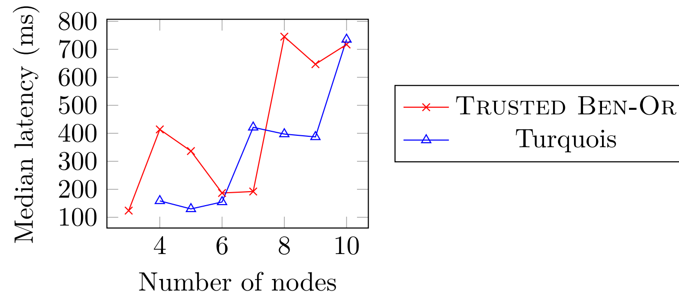
(b) Box plot of latency. The three horizontal bars of the box represent the first, second (median) and third quantiles respectively; the whiskers correspond to the 9th and 91st percentiles respectively.

Figure 1 Latency of consensus in fault-free case.

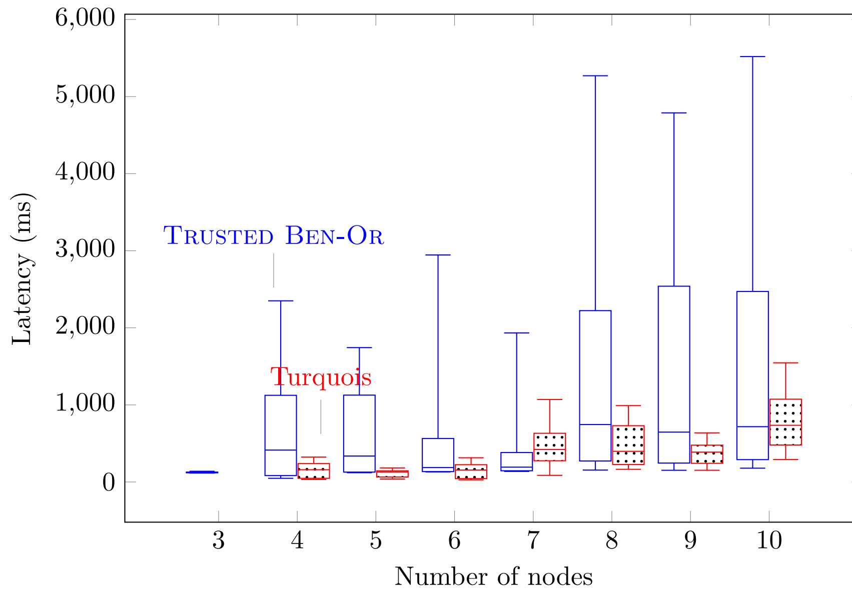
communicate via multicast broadcast. As a result, the nodes are more likely to receive the same vector of proposals, then pick the common value for the third phase. On the other hand, Trusted Ben-Or has only two phases in each round. If they propose different values, very likely they have to toss the coin to start the next round. In fact, we have seen more than 50 rounds until they all decide in a group with 10 nodes.

7 Related Work

The impossibility result by Fischer, Lynch and Paterson [13] precludes any deterministic consensus algorithm in asynchronous systems with even a single faulty process. There are two approaches to bypass this impossibility. Firstly we can resort to a weakened system model, e. g. a partially synchronous model [12]. The other option is to use a randomized algorithm (although the correctness criterion is correspondingly weakened from “must terminate” to “terminates with a probability of 1”).



(a) Median of latency.



(b) Box plot of latency. The meaning of the boxes is the same as Figure 1b.

Figure 2 Latency of consensus when Byzantine processes exist.

Ben-Or's algorithm [6] is a randomized fault tolerant consensus algorithm for a completely asynchronous system and can withstand a strong adversary. It has a crash fault tolerant variant and a BFT variant. The former can tolerate $f \leq \lfloor \frac{n-1}{2} \rfloor$ crashed processes, which has inspired this work. The BFT version requires $n > 5f$, which is less attractive in practice. Bracha's algorithm [7] improves the maximum tolerable faults to $f \leq \lfloor \frac{n-1}{3} \rfloor$ at the cost of using reliable broadcast, which can introduce considerable overhead. Turquois [19] has a novel message validation mechanism to get rid of the reliable broadcast primitive. Meanwhile, it utilizes an efficient message authentication approach and UDP broadcast, making it tailored for wireless embedded systems. The authors did not mention the strong/weak adversary model, but it turns out that Turquois cannot withstand a strong adversary, as discussed in Section 5.1. Several works have explicitly addressed the weak adversary model [9, 1], in which the adversary does not know everything about the whole system state. They achieve high efficiency at the cost of having this weakened adversary model. Vavala and Neves also propose a speculative randomized consensus algorithm in a so-called *normal condition* where the adversary model is further relaxed compared to the worst case [20]. It is worth mentioning the correctness proof of the crash fault tolerant Ben-Or's algorithm [2]. It gives

a good example of how to proof termination under a strong adversary model. But if we take Byzantine faults into account, the proof becomes more complex as we show in this work.

The hybrid fault model [21] is a common approach to increase the maximum tolerable faulty processes and to decrease the complexity of consensus. In this model, a small subset of the system is trusted and cannot be arbitrarily faulty, but can only fail by crashing. One common usage of this subsystem is to prevent equivocation by using one or more monotonic counters for message authentication [17, 22, 15, 5, 23], so that a Byzantine process cannot send contradictory messages to different recipients — similar to the identical Byzantine fault model [4]. Among them, Ratcheta [23] is also tailored for wireless embedded systems. But all the above mentioned works are deterministic algorithms and assume partial synchrony. Correia et al. [10] discuss the transformation from a crash consensus to Byzantine consensus in asynchronous systems by means of using the hybrid fault model. Although they provide an idea to transform the original Ben-Or's algorithm, above all they require the reliable broadcast primitive. Besides that, the algorithm relies on a failure detector that can eventually find out all Byzantine processes, but the design of this failure detector is unclear. Moreover, they do not mention a trusted random number generator, so the termination seems problematic.

8 Conclusion

In this work, we present Trusted Ben-Or, a randomized hybrid fault-tolerant consensus algorithm. It is operational in an asynchronous system and is resilient against a strong adversary, as long as the adversary cannot compromise the trusted subsystem of each process. Trusted Ben-Or increases the maximum tolerable Byzantine processes to $\lfloor\frac{n-1}{2}\rfloor$. The algorithm is tailored for wireless embedded systems because it does not rely on connection-oriented communication protocols, e. g. TCP, or any complex communication primitives. Neither does it require expensive asymmetric digital signatures. We evaluate Trusted Ben-Or on a testbed consisting of 10 Raspberry Pis connected via an ad hoc wireless network. The results show that the median latency is below 250 ms, and for most of the time Trusted Ben-Or achieves almost the same performance as Turquois – another well-known asynchronous BFT consensus algorithm tolerating only up to $\lfloor\frac{n-1}{3}\rfloor$ Byzantine processes.

References

1. Ittai Abraham, Danny Dolev, and Joseph Y Halpern. An almost-surely terminating polynomial protocol for asynchronous byzantine agreement with optimal resilience. In *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, pages 405–414. ACM, 2008.
2. Marcos K Aguilera and Sam Toueg. The correctness proof of Ben-Or's randomized consensus algorithm. *Distributed Computing*, 25(5):371–381, 2012.
3. ARM. ARM Security Technology - Building a Secure System using TrustZone Technology, ARM Technical White Paper, 2009.
4. H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*, pages 255–256. Wiley Series on Parallel and Distributed Computing. Wiley, 2004.
5. Johannes Behl, Tobias Distler, and Rüdiger Kapitza. Hybrids on Steroids: SGX-Based High Performance BFT. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 222–237. ACM, 2017.
6. Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 27–30. ACM, 1983.

7 Gabriel Bracha. An asynchronous [(n-1)/3]-resilient consensus protocol. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 154–162. ACM, 1984.

8 Gabriel Bracha and Sam Toueg. Resilient consensus protocols. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 12–26. ACM, 1983.

9 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*, pages 42–51. ACM, 1993.

10 Miguel Correia, Giuliana S Veronese, and Lau Cheuk Lung. Asynchronous Byzantine consensus with 2f+ 1 processes. In *Proceedings of the 2010 ACM symposium on applied computing*, pages 475–480. ACM, 2010.

11 J. Q. Cui, S. K. Phang, K. Z. Y. Ang, F. Wang, X. Dong, Y. Ke, S. Lai, K. Li, X. Li, F. Lin, J. Lin, P. Liu, T. Pang, B. Wang, K. Wang, Z. Yang, and B. M. Chen. Drones for cooperative search and rescue in post-disaster situation. In *2015 IEEE 7th International Conference on Cybernetics and Intelligent Systems (CIS) and IEEE Conference on Robotics, Automation and Mechatronics (RAM)*, pages 167–174, 2015. doi:10.1109/ICCIS.2015.7274615.

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

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

14 Zafar Iqbal, Kiseon Kim, and Heung-No Lee. A cooperative wireless sensor network for indoor industrial monitoring. *IEEE Transactions on Industrial Informatics*, 13(2):482–491, 2017.

15 Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. CheapBFT: Resource-efficient Byzantine Fault Tolerance. In European Chapter of ACM SIGOPS, editor, *Proceedings of the EuroSys 2012 Conference*, pages 295–308, 2012.

16 Yasuhiro Kuriki and Toru Namerikawa. Consensus-based cooperative formation control with collision avoidance for a multi-UAV system. In *American Control Conference (ACC), 2014*, pages 2077–2082. IEEE, 2014.

17 Dave Levin, John R Douceur, Jacob R Lorch, and Thomas Moscibroda. TrInc: Small Trusted Hardware for Large Distributed Systems. In *NSDI*, volume 9, pages 1–14, 2009.

18 Linaro. Open Portable Trusted Execution Environment. https://www.op-tee.org, visited in September, 2018.

19 Henrique Moniz, Nuno Ferreira Neves, and Miguel Correia. Turquois: Byzantine consensus in wireless ad hoc networks. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pages 537–546. IEEE, 2010.

20 Bruno Vavala and Nuno Neves. Robust and speculative Byzantine randomized consensus with constant time complexity in normal conditions. In *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on*, pages 161–170. IEEE, 2012.

21 Paulo E Veríssimo. Travelling through wormholes: a new look at distributed systems models. *ACM SIGACT News*, 37(1):66–81, 2006.

22 Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. Efficient byzantine fault-tolerance. *Computers, IEEE Transactions on*, 62(1):16–30, 2013.

23 Wenbo Xu and Rüdiger Kapitza. RATCHETA: Memory-bounded Hybrid Byzantine Consensus for Cooperative Embedded Systems. In *Reliable Distributed Systems (SRDS), 2018 IEEE thirty-seventh Symposium on*. IEEE, 2018.

Correctness of Tendermint-Core Blockchains

Yackolley Amoussou-Guenou
Institut LIST, CEA, Université Paris-Saclay, F-91120, Palaiseau, France
Sorbonne Université, CNRS, Laboratoire d'Informatique de Paris 6, F-75005 Paris, France

Antonella Del Pozzo
Institut LIST, CEA, Université Paris-Saclay, F-91120, Palaiseau, France

Maria Potop-Butucaru
Sorbonne Université, CNRS, Laboratoire d'Informatique de Paris 6, F-75005 Paris, France

Sara Tucci-Piergiovanni
Institut LIST, CEA, Université Paris-Saclay, F-91120, Palaiseau, France

Abstract

Tendermint-core blockchains (e.g. Cosmos) are considered today one of the most viable alternatives for the highly energy consuming proof-of-work blockchains such as Bitcoin and Ethereum. Their particularity is that they aim at offering strong consistency (no forks) in an open system combining two ingredients (i) a set of validators that generate blocks via a variant of Practical Byzantine Fault Tolerant (PBFT) consensus protocol and (ii) a selection strategy that dynamically selects nodes to be validators for the next block via a proof-of-stake mechanism. The exact assumptions on the system model under which Tendermint underlying algorithms are correct and the exact properties Tendermint verifies, however, have never been formally analyzed. The contribution of this paper is as follows. First, while formalizing Tendermint algorithms we precisely characterize the system model and the exact problem solved by Tendermint, then, we prove that in eventual synchronous systems a modified version of Tendermint solves (i) under additional assumptions, a variant of one-shot consensus for the validation of one single block and (ii) a variant of the repeated consensus problem for multiple blocks. These results hold even if the set of validators is hit by Byzantine failures, provided that for each one-shot consensus instance less than one third of the validators is Byzantine.

2012 ACM Subject Classification Computer systems organization → Dependable and fault-tolerant systems and networks

Keywords and phrases Blockchain, Consensus, Proof-of-Stake, Fairness

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.16

Related Version A full version is available at `https://eprint.iacr.org/2018/574.pdf`.

Acknowledgements The authors thank anonymous reviewers for their insightful comments.

1 Introduction

Blockchain is today one of the most appealing technology since its introduction in the Bitcoin White Paper [30] in 2008. Blockchain systems, similar to P2P systems in the early 2000, take their roots in the non academic research. After the releasing of the most popular blockchains (e.g. Bitcoin [30] or Ethereum [36]) with a specific focus on economical transactions, their huge potential for various other applications ranging from notary to medical data recording became evident. In a nutshell, Blockchain systems maintain a continuously-growing history of ordered information, encapsulated in blocks. Blocks are linked to each other by relying on

© Yackolley Amoussou-Guenou, and Antonella Del Pozzo, and Maria Potop-Butucaru, and Sara Tucci-Piergiovanni;
licensed under Creative Commons License CC-BY
22nd International Conference on Principles of Distributed Systems (OPODIS 2018).
Editors: Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira; Article No. 16; pp. 16:1–16:16

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

collision resistant hash functions, i.e., each block contains the hash of the previous block. The Blockchain itself is a distributed data structure replicated among different peers. In order to preserve the chain structure those peers need to agree on the next block to append in order to avoid forks. The most popular technique to decide which block will be appended is the *proof-of-work* mechanism of Dwork and Naor [15]. The block that will be appended to the blockchain is owned by the node (miner) having enough CPU power to solve first a crypto-puzzle first. The only possible way to solve this puzzle is by repeated trials. The major criticisms for the *proof-of-work* approach are as follows: it is assumed that the honest miners hold a majority of the computational power, the generation of a block is energetically costly, which yield to the creation of mining pools and finally, multiple blockchains might coexist in the system due to accidental or intentional forks.

Recently, the non academic research developed alternative solutions to the proof-of-work technique such as *proof-of-stake* (the power of block building is proportional to the participant wealth), *proof-of-space* (similar to proof-of-work, instead of CPU power the prover has to provide the evidence of a certain amount of space) or *proof-of-authority* (the power of block building is proportional to the amount of authority owned in the system). These alternatives received little attention in the academic research. Among all these alternatives *proof-of-stake* protocols and in particular those using variants of *Practical Byzantine Fault-Tolerant* consensus [8] became recently popular not only for in-chain transaction systems but also in systems that provide cross-chain transactions. Tendermint [27, 7, 25, 28] was the first in this line of research having the merit to link the *Practical Byzantine Fault-Tolerant* consensus to the proof-of-stake technique and to propose a blockchain where a dynamic set of validators (subset of the participants) decide on the next block to be appended to the blockchain. Although, the correctness of the original Tendermint protocol [27, 7, 25] has never been formally analyzed from the distributed computing perspective, it or slightly modified variants became recently the core of several popular systems such as Cosmos [26] for cross-chain transactions.

In this paper we analyse the correctness of the original Tendermint agreement protocol as it was described in [27, 7, 25] and discussed in [28, 22]. The code of this protocol is available in [35]. One of our fundamental results proved in this paper is as follows:

In an eventual synchronous system, a slightly modified variant of the original Tendermint protocol implements the one-shot and repeated consensus, provided that (i) the number of Byzantine validators, f, is $f < n/3$ where n is the number of validators participating in each single one-shot consensus instance and (ii) eventually a proposed value will be accepted by at least $2n/3 + 1$ processes (Theorem 7 and Theorem 8).

More in detail, we prove that the original Tendermint (specified for the first time in a preliminary version of this work, see technical report [2]) verifies the consensus termination with a small twist in the algorithm (a refinement of the timeout) and with the additional assumption stating that there exists eventually a proposer such that its proposed value will be accepted, or voted, by more than two-third of validators. The rest of the paper is organized as follows. Related works are discussed in Section 2. Section 3 defines the model and the formal specifications of one-shot and repeated consensus. Section 4 formalizes the original Tendermint One-Shot and Repeated Consensus protocols through pseudo-code and proves the correctness of the One-Shot Consensus algorithm. Due to space limitations, the fairness study along with full descriptions of the counter-examples that motivate the modification of the original algorithm and the additional assumptions for correctness are provided in [2].

2 Related Work

Interestingly, only recently distributed computing academic scholars focus their attention on the theoretical aspects of blockchains motivated mainly by the intriguing claim of popular blockchains, as Bitcoin and Ethereum, that they implement consensus in an asynchronous dynamic open system. This claim is refuted by the famous impossibility result in distributing computing [17]. In distributed systems, the theoretical studies of *proof-of-work* based blockchains have been pioneered by Garay et *al* [19]. Garay *et al.* decorticate the pseudo-code of Bitcoin and analyse its agreement aspects considering a synchronous round-based communication model. This study has been extended by Pass *et al.* [31] to round based systems where messages sent in a round can be received later. [16] proposes a mix between proof-of-work blockchains and proof-of-work free blockchains referred as Bitcoin-NG. Bitcoin-NG inherits, however, the drawbacks of Bitcoin: costly proof-of-work process, forks, no guarantee that a leader in an epoch is unique, no guarantee that the leader does not change the history at will if it is corrupted. On another line of research, in [11] Decker *et al.* propose the PeerCensus system that targets linearizability of transactions. PeerCensus combines the proof-of-work blockchain and the classical results in Practical Byzantine Fault Tolerant agreement area. PeerCensus suffers the same drawbacks as Bitcoin because of the proof-of-work. Byzcoin [24] builds on top of *Practical Byzantine Fault-Tolerant* consensus [8] enhanced with a scalable collective signing process. [24] is based on a leader-based consensus over a group of members chosen by a proof-of-membership mechanism. When a miner succeeds to mine a block, it gains a membership share, and the miners with the highest shares are part of the fixed size voting member set. In the same spirit, SBFT [21] and Hyperledger Fabric [3] build on top of [8]. In [32] and [20], *sortition* based blockchains are discussed, where the proof-of-work mechanism is completely replaced by a probabilistic ingredient.

The only academic works that address the consensus in *proof-of-stake* based blockchains are [10] and [23]. In [10], Daian *et al.* which proposes a protocol for weakly synchronous networks. The execution of the protocol is organized in epochs. Similar to Bitcoin-NG [16] in each epoch a different committee is elected and inside the elected committee a leader will be chosen. The leader is allowed to extend the new blockchain. The protocol is validated via simulations and only partial proofs of correctness are provided. Ouroboros [23] proposes a sortition based proof-of-stake protocol and addresses mainly the security aspects of the proposed protocol. Red Belly [9] focuses on consortium blockchains, where only a predefined subset of processes are allowed to update the blockchain, and proposes a Byzantine consensus protocol.

Interestingly, none of the previous academic studies made the connection between the repeated consensus specification [5, 13, 12] and the repeated agreement process in blockchain systems. Moreover, in terms of fairness of rewards, no academic study has been conducted related to blockchains based on repeated consensus.

3 System model and Problem Definition

The system is composed of an infinite set Π of asynchronous sequential processes, namely $\Pi = \{p_1, \dots\}$; i is called the *index* of p_i. *Asynchronous* means that each process proceeds at it own speed, which can vary with time and remains unknown to the other processes. *Sequential* means that a process executes one step at a time. This does not prevent it from executing several threads with an appropriate multiplexing. As local processing times are negligible with respect to message transfer delays, they are considered as being equal to zero.

Arrival model. We assume a *finite arrival model* [1], i.e. the system has infinitely many processes but each run has only finitely many. The size of the set $\Pi_\rho \subset \Pi$ of processes that participate in each system run is not a priori-known. We also consider a finite subset $V \subseteq \Pi_\rho$ of validators. The set V may change during any system run and its size n is a-priori known. A process is promoted in V based on a so-called merit parameter, which can model for instance its stake in proof-of-stake blockchains. Note that in the current Tendermint implementation, it is a separate module included in the Cosmos project [26] that is in charge of implementing the selection of V.

Communication network. The processes communicate by exchanging messages through an eventually synchronous network [14]. *Eventually Synchronous* means that after a finite unknown time τ there is an upper bound δ on the message transfer delay.

Failure model. There is no bound on processes that can exhibit a Byzantine behaviour [33] in the system, but up to f validators can exhibit a Byzantine behaviour at each point of the execution. A Byzantine process is a process that behaves arbitrarily: it can crash, fail to send or receive messages, send arbitrary messages, start in an arbitrary state, perform arbitrary state transitions, etc. Byzantine processes can control the network by modifying the order in which messages are received, but they cannot postpone forever message receptions. Moreover, Byzantine processes can collude to "pollute" the computation (e.g., by sending messages with different contents, while they should send messages with the same content if they were non-faulty). A process (or validator) that exhibits a Byzantine behaviour is called *faulty*. Otherwise, it is *non-faulty* or *correct*. To be able to solve the consensus problem, we assume that $f < n/3$.

Communication primitives. In the following we assume the presence of a broadcast primitive. A process p_i broadcasts a message by invoking the primitive $\mathsf{broadcast}(\langle TAG, m\rangle)$, where TAG is the type of the message, and m its content. To simplify the presentation, it is assumed that a process can send messages to itself. The primitive $\mathsf{broadcast}()$ is a best effort broadcast, which means that when a correct process broadcasts a value, eventually all the correct processes deliver it. A process p_i receives a message by executing the primitive $\mathsf{delivery}()$. Messages are created with a digital signature, and we assume that digital signatures cannot be forged. When a process p_i delivers a message, it knows the process p_j that created the message.

Let us note that the assumed broadcast primitive in an open dynamic network can be implemented through *gossiping*, i.e. each process sends the message to current neighbors in the underlying dynamic network graph. In these settings the finite arrival model is a necessary condition for the system to show eventual synchrony. Intuitively, a finite arrival implies that message losses due to topology changes are bounded, so that the propagation delay of a message between two processes not directly connected can be bounded [29, 4].

Problem definition. In this paper we analyse the correctness of Tendermint protocol against two abstractions in distributed systems: one-shot consensus and repeated consensus defined formally as follows.

▶ **Definition 1** (One-Shot Consensus)**.** We say that an algorithm implements One-Shot Consensus if and only if it satisfies the following properties:

- **Termination.** Every correct process eventually decides some value.
- **Integrity.** No correct process decides twice.

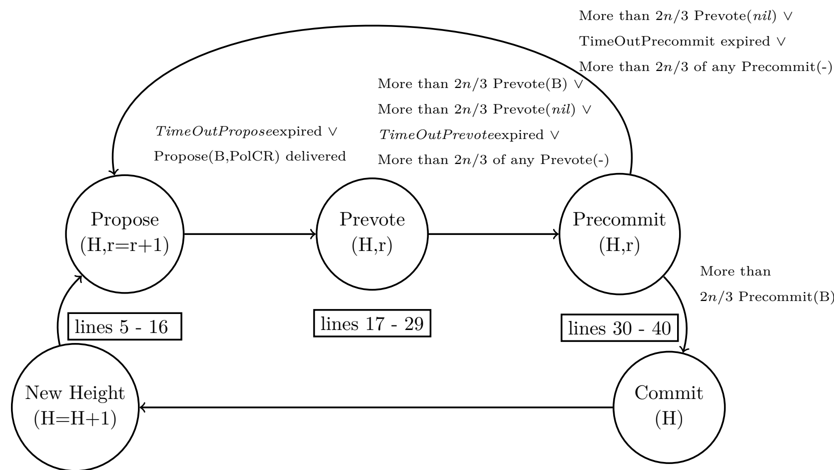


Figure 1 State Machine for Tendermint One-Shot algorithm described in Figure 3.

- **Agreement.** If there is a correct process that decides a value B, then eventually all the correct processes decide B.
- **Validity[9].** A decided value is valid, it satisfies the predefined predicate denoted isValid().

The concept of multi-consensus is presented in [5], where the authors assume that only the faulty processes can postpone the decision of correct processes. In addition, the consensus is made a finite number of times. The long-lived consensus presented in [13] studies the consensus when the inputs are changing over the time, their specification aims at studying in which condition the decisions of correct process do not change over time. None of these specifications is appropriate for blockchain systems. In [12], Delporte-Gallet *et al.* defined the Repeated Consensus as an infinite sequence of One-Shot Consensus instances, where the inputs values may be completely different from one instance to another, but where all the correct processes have the same infinite sequence of decisions. We consider a variant of the repeated consensus problem as defined in [12]. The main difference is that we do not predicate on the faulty processes. Each correct process outputs an infinite sequence of decisions. We call that sequence the *output* of the process.

▶ **Definition 2** (Repeated Consensus)**.** An algorithm implements a repeated consensus if and only if it satisfies the following properties:

- **Termination.** Every correct process has an infinite output.
- **Agreement.** If the ith value of the output of a correct process is B, then B is the ith value of the output of any other correct process.
- **Validity.** Each value in the output of any correct process is valid, it satisfies the predefined predicate denoted isValid().

4 Tendermint Formalization

4.1 Informal description of Tendermint and its blockchain

Tendermint protocol [27, 7] aims at building a blockchain without forks relying on a variant of PBFT consensus. When building the blockchain, a subset of fixed size n of processes called

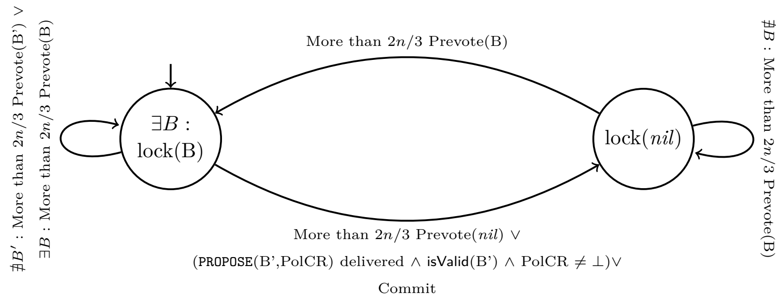


Figure 2 State machine Lock/Unlock

validators should agree on the next block to append to the blockchain. The set of validators is deterministically determined by the current content of the blockchain, referred as the history. We note that this subset may change once a block is appended. The mechanism to choose the validators from a given history is further referred as *selection mechanism*. Note that in the current Tendermint implementation, it is a separate module included in the Cosmos project [26] that is in charge of implementing the selection mechanism. Intuitively, such mechanism should be based on the proof-of-stake approach but its actual implementation is currently left open.

The first block of Tendermint blockchain, called the *genesis block*, is at *height* 0. The *height* of a block is the distance that separates that block to the genesis block. Each block contains: (i) a *Header* which contains a pointer to the previous block and the height of the block, (ii) the *Data* which is a list of transactions, and (iii) a set *LastCommit* which is the set of validators that signed on the previous block. Except the first block, each block refers to the previous block in the chain. Given a current height of Tendermint blockchain, a total ordered set of validators V is selected to add a new block. The validators start a *One-Shot Consensus algorithm*. The first validator creates and proposes a block B, then if more than $2n/3$ of the validators accept B, B will be appended as the next block, otherwise the next validator proposes a block, and the mechanism is repeated until more than $2n/3$ of the validators accept a block. For each height of Tendermint blockchain, the mechanism to append a new block is the same, only the set of validators may change. Therefore, Tendermint applies a *Repeated Consensus algorithm* to build a blockchain, and at each height, it relies on a *One-Shot Consensus algorithm* to decide the block to be appended. Due to space limitations, Repeated Consensus proofs are provided in [2].

Although the choice of validators is managed by a separate module (see Cosmos project [26]) the rewards for the validators that contributed to the block at some specific height H are determined during the construction of the block at height $H+1$. The validators for H that get a reward for H are the ones that validators for $H+1$ "saw" when proposing a block. This mechanism can be unfair, since some validator for H may be slow, and its messages may not reach the validators involved in $H+1$, implying that it may not get the rewards it deserved. Due to space limitations, our fairness study are provided in [2].

4.2 Tendermint One-Shot Consensus algorithm

Tendermint One-Shot Consensus algorithm is a round-based algorithm used to decide on the next block for a given height H. In each *round* there is a different proposer that proposes a block to the validators that try to decide on that block. A round consists of three steps: (i) the *Propose step*, the proposer of the round broadcasts a proposal for a block; (ii) the *Prevote*

Function consensus$(H, \Pi_\rho, \text{signature})$; %One-Shot Consensus for the height H with the set Π_ρ of processes%

Init:
(1) $r \leftarrow 0$; $LLR_i \leftarrow -1$; $PoLCR_i \leftarrow \perp$; $lockedBlock_i \leftarrow nil$; $B \leftarrow nil$;
(2) $TimeOutPropose \leftarrow \Delta_{\text{Propose}}$; $TimeOutPrevote \leftarrow \Delta_{\text{Prevote}}$;
(3) $proposalReceived_i^{H,r} \leftarrow \perp$; $prevotesReceived_i^{H,r} \leftarrow \perp$; $precommitsReceived_i^{H,r} \leftarrow \perp$;

---

while (true) **do**
(4) $r \leftarrow r + 1$; $PoLCR_i \leftarrow \perp$;
——— Propose step r ———
(5) **if** $(p_i == \text{proposer}(H, r))$ **then**
(6) **if** $(LLR_i \neq -1)$ **then** $PoLCR_i \leftarrow LLR_i$; $B \leftarrow lockedBlock_i$;
(7) **else** $B \leftarrow$ createNewBlock(signature);
(8) **endif**
(9) **trigger** broadcast $\langle$PROPOSE, $(B, H, r, PoLCR_i)_i\rangle$;
(10) **else**
(11) set $timerProposer$ to $TimeOutPropose$;
(12) **wait until** $((timerProposer \text{ expired}) \vee (proposalReceived_i^{H,r'} \neq \perp))$;
(13) **if** $((timerProposer \text{ expired}) \wedge (proposalReceived_i^{H,r'} == \perp))$ **then**
(14) $TimeOutPropose \leftarrow TimeOutPropose + 1$;
(15) **endif**
(16) **endif**
——— Prevote step r ———
(17) **if** $((PoLCR_i \neq \perp) \wedge (LLR_i \neq -1) \wedge (LLR_i < PoLCR_i < r))$ **then**
(18) **wait until** $|prevotesReceived_i^{H,PoLCR}| > 2n/3$;
(19) **if** $(\exists B' : (\text{is23Maj}(B', prevotesReceived_i^{H,PoLCR_i})) \wedge (B' \neq lockedBlock_i))$ **then** $lockedBlock_i \leftarrow nil$; **endif**
(20) **endif**
(21) **if** $(lockedBlock_i \neq nil)$ **then trigger** broadcast $\langle$PREVOTE, $(lockedBlock_i, H, r)_i\rangle$;
(22) **else if** $(\text{isValid}(proposalReceived_i^{H,r}))$ **then trigger** broadcast $\langle$PREVOTE, $(proposalReceived_i^{H,r}, H, r)_i\rangle$; **endif**
(23) **else trigger** broadcast $\langle$PREVOTE, $(nil, H, r)_i\rangle$;
(24) **endif**
(25) **wait until** $((\text{is23Maj}(nil, prevotesReceived_i^{H,r})) \vee (\exists B'' : (\text{is23Maj}(B'', prevotesReceived_i^{H,r}))) \vee$
$(|prevotesReceived_i^{H,r}| > 2n/3))$; %Delivery of any 2n/3 prevotes for the round r%
(26) **if** $(\neg(\text{is23Maj}(nil, prevotesReceived_i^{H,r})) \wedge \neg(\exists B'' : (\text{is23Maj}(B'', prevotesReceived_i^{H,r}))))$ **then**
(27) **set** $timerPrevote$ **to** $TimeOutPrevote$;
(28) **wait until** $(timerPrevote$ expired$)$;
(29) **if** $(timerPrevote$ expired$)$ **then** $TimeOutPrevote \leftarrow TimeOutPrevote + 1$; **endif**
——— Precommit step r ———
(30) **if** $(\exists B' : (\text{is23Maj}(B', prevotesReceived_i^{H,r})))$ **then**
(31) $lockedBlock_i \leftarrow B'$;
(32) **trigger** broadcast $\langle$PRECOMMIT, $(B', H, r)_i\rangle$;
(33) $LLR_i \leftarrow r$;
(34) **else if** $(\text{is23Maj}(nil, prevotesReceived_i^{H,r}))$ **then**
(35) $lockedBlock_i \leftarrow nil$; $LLR_i \leftarrow -1$;
(36) **trigger** broadcast $\langle$PRECOMMIT, $(nil, H, r)_i\rangle$;
(37) **endif**
(38) **else trigger** broadcast $\langle$PRECOMMIT, $(nil, H, r)_i\rangle)$;
(39) **endif**
(40) **wait until** $((\text{is23Maj}(nil, prevotesReceived_i^{H,r})) \vee (|precommitsReceived_i^{H,r}| > 2n/3))$
endwhile

Figure 3 First part of Tendermint One-shot Consensus algorithm at correct process p_i.

step, validators broadcast their prevotes depending on the proposal they delivered during the previous step; and (iii) the *Precommit step*, validators broadcast their precommits depending on the occurrences of prevotes for the same block they delivered during the previous step. To preserve liveness, steps have a timeout associated, so that each validator moves from one step to another either if the timeout expires or if it delivers enough messages of a particular typology. When p_i broadcasts a message $(\langle TAG, m\rangle)$, m contains a block B along with other information. We say that p_i prevotes (resp. precommits) on B if $TAG =$ PREVOTE (resp. TAG=PRECOMMIT). In Figure 1 is depicted the state machine for the Tendermint one-Shot Consensus.

To preserve safety of the protocol and to satisfy the Agreement property, when a validator delivers more than $2n/3$ prevotes for B then it "locks" on such block. Informally, it means that there are at least $n/3 + 1$ prevotes for B from correct processes, then B is a possible candidate for a decision so that validators try to stick on that. More formally, a validator

```
upon event delivery ⟨PROPOSE, (B', H, r', PoLCR_j)_j⟩:
(41) if (proposalReceived_i^{H,r'} = ⊥) then
(42)   proposalReceived_i^{H,r'} ← (B', H, r')_j;
(43)   PoLCR_i ← PoLCR_j;
(44)   trigger broadcast ⟨PROPOSE, (B', H, r', PoLCR_j)_j⟩;
(45) endif
------------------------------------------------------------
upon event delivery ⟨PREVOTE, (B', H, r', LLR)_j⟩:
(46) if ((B', H, r', LLR)_j ∉ prevotesReceived_i^{H,r'}) then
(47)   prevotesReceived_i^{H,r'} ← prevotesReceived_i^{H,r'} ∪ (B', H, r', LLR)_j;
(48)   trigger broadcast ⟨PREVOTE, (B', H, r', LLR)_j⟩;
(49)   if ((r < r') and (|prevotesReceived_i^{H,r'}| > 2/3)) then
(50)     r ← r';
(51)     goto Prevote step r;
(52)   endif
(53) endif
------------------------------------------------------------
upon event delivery ⟨PRECOMMIT, (B', H, r')_j⟩:
(54) if ((B', H, r')_j ∉ precommitsReceived_i^{H,r'}) then
(55)   precommitsReceived_i^{H,r'} ← precommitsReceived_i^{H,r'} ∪ (B', H, r')_j;
(56)   trigger broadcast ⟨PRECOMMIT, (B', H, r')_j⟩;
(57)   if ((r < r') and (|precommitsReceived_i^{H,r'}| > 2/3)) then
(58)     r ← r';
(59)     goto Precommit step r;
(60)   endif
(61) endif
------------------------------------------------------------
when (∃B' : is23Maj(B', precommitsReceived_i^{H,r'})):
(62) return B';%Terminate the consensus for the height H by deciding B'%
```

Figure 3 Second part of Tendermint One-shot Consensus algorithm at correct process p_i.

has a *Proof-of-LoCk* (PoLC) for a block B (resp. for *nil*) at a round r for the height H if it received at least $2n/3+1$ prevotes for B (resp. for *nil*). In this case we say that a process is locked on such block. A *PoLC-Round* (*PoLCR*) is a round such that there was a PoLC for a block at round *PoLCR*. In Figure 2 the state machine concerning the process of locking and unlocking on a block B is shown.

Preamble. Note that our analysis of the original Tendermint protocol [27, 7, 25] led to the conclusion that several modifications were needed in order to implement One-Shot Consensus problem. Full description of these bugs in the original Tendermint protocol are reported in [2]. In more details, with respect to the original Tendermint, our Tendermint One-shot Consensus algorithm (see Figure 3) has the following modifications. We added line 29 in order to catch up the communication delay during the synchronous periods. Moreover, we modified the line 19 in order to guarantee the agreement property of One-Shot Consensus (defined in Section 3). The correctness of Tendermint One-shot Consensus algorithm needs an additional assumption stating that eventually a proposal is accepted by a majority of correct processes. This assumption, stated formally in Theorem 7, is necessary to guarantee the termination.

Variables and data structures. r and $PoLCR_i$ are integers representing, respectively, the current round and the PolCR. $lockedBlock_i$ is the last block on which p_i is locked, if it is equal to a block B, we say that p_i is locked on B, otherwise it is equal to *nil*, and we say that p_i is not locked. When $lockedBlock_i \neq nil$ and switches the value to *nil*, then p_i unlocks. Last-Locked-Round (LLR_i) is an integer representing the last round where p_i locked on a block. B is the block the process created.

Each validator manages timeouts, *TimeOutPropose* and *TimeOutPrevote*, concerning the propose and prevote phases respectively. Those timeouts are set to Δ_{Propose} and Δ_{Prevote} and are started at the beginning of the respective step. Both are incremented if they expire before the validator moves to the next step.

Each validator manages three sets for the messages delivered. In particular, the set $proposalReceived_i^{H,r}$ contains the proposal that p_i delivered for the round r at height H. $prevotesReceived_i^{H,r}$ is the set containing all the prevotes p_i delivered for the round r at height H. $precommitsReceived_i^{H,r}$ is the set containing all the precommits p_i delivered for the round r at height H.

Functions. We denote by *Block* the set containing all blocks, and by *MemPool* the structure containing all the transactions.

- proposer : $V \times Height \times Round \to V$ is a deterministic function which gives the proposer out of the validators for a given round at a given height in a round robin fashion.
- createNewBlock : $2^{\Pi_\rho} \times MemPool \to Block$ is an application-dependent function which creates a valid block (w.r.t. the application), where the subset of processes is a parameter of the One-Shot Consensus, and is a subset of processes that send a commit for the block at the previous height, called the signature of the previous block.
- is23Maj : $(Block \cup nil) \times (prevotesReceived \cup precommitsReceived) \to$ Bool is a predicate that checks if there is at least $2n/3+1$ of prevotes or precommits on the given block or *nil* in the given set.
- isValid : $Block \to$ Bool is an application dependent predicate that is satisfied if the given block is valid. If there is a block B such that isValid$(B) =$ true, we say that B is valid. We note that for any non-block, we set isValid to false, (e.g. isValid$(nil) =$ false).

Detailed description of the algorithm. In Figure 3 we describe Tendermint One-Shot algorithm to solve the One-Shot Consensus (defined in Section 3) for a given height H.
For each round r at height H the algorithm proceeds in 3 phases:

1. Propose step (lines 5 - 16): If p_i is the proposer of the round and it is not locked on any block, then it creates a valid proposal and broadcasts it. Otherwise it broadcasts the block it is locked on. If p_i is not the proposer then it waits for the proposal from the proposer. p_i sets the timer to *TimeOutProposal*, if the timer expires before the delivery of the proposal then p_i increases the time-out, otherwise it stores the proposal in $proposalReceived_i^{H,r}$. In any case, p_i goes to the Prevote step.
2. Prevote step (lines 17 - 29): If p_i delivers the proposal during the Propose step, then it checks the data on the proposal. If $lockedBlock_i \neq nil$, and p_i delivers a proposal with a valid *PoLCR* then it unlocks. After that check, if p_i is still locked on a block, then it prevotes on $lockedBlock_i$; otherwise it checks if the block B in the proposal is valid or not, if B is valid, then it prevotes B, otherwise it prevotes on *nil*. Then p_i waits until $|prevotesReceived_i^{H,r}| > 2n/3$. If there is no PoLC for a block or for *nil* for the round r, then p_i sets the timer to *TimeOutPrevote*, waits for the timer's expiration and increases *TimeOutPrevote*. In any case, p_i goes to Precommit step.
3. Precommit step (lines 30 - 40): p_i checks if there was a PolC for a particular block or *nil* during the round (lines 30 and 34). There are three cases: (i) if there is a PoLC for a block B, then it locks on B, and precommits on B (lines 30 - 32); (ii) if there is a PoLC for *nil*, then it unlocks and precommits on *nil* (lines 34 - 36); (iii) otherwise, it precommits on *nil* (line 38); in any case, p_i waits until $|precommitsReceived_i^{H,r}| > 2n/3$ or (is23Maj$(nil, prevotesReceived_i^{H,r})$), and it goes to the next round.

Whenever p_i delivers a message, it broadcasts it (lines 44, 48 and 56). Moreover, during a round r, some conditions may be verified after a delivery of some messages and either (i) p_i decides and terminates or (ii) p_i goes to the round r' (with $r' > r$). The conditions are:

- For any round r', if for a block B, $\mathsf{is23Maj}(B, \mathit{precommitsReceived}_i^{H,r'}) = \mathsf{true}$, then p_i decides the block B and terminates, or
- If p_i is in the round r at height H and $|\mathit{prevotesReceived}_i^{H,r'}| > 2n/3$ where $r' > r$, then it goes to the Prevote step for the round r', or
- If p_i is in the round r at height H and $|\mathit{precommitsReceived}_i^{H,r'}| > 2n/3$ where $r' > r$, then it goes to the Precommit step for the round r'.

4.2.1 Correctness of Tendermint One-Shot Consensus

In this section we prove the correctness of Tendermint One-Shot Consensus algorithm (Fig. 3) for a height H under the assumption that during the synchronous period there exists eventually a proposer such that its proposed value will be accepted by at least $2n/3 + 1$ processes.

▶ **Lemma 1** (One-Shot Integrity). *In an eventual synchronous system, Tendermint One-Shot Consensus Algorithm verifies the following property: No correct process decides twice.*

▶ **Lemma 2** (One-Shot Validity). *In an eventual synchronous system, Tendermint One-Shot Consensus Algorithm verifies the following property: A decided value is valid, if it satisfies the predefined predicate denoted isValid().*

▶ **Lemma 3.** *In an eventual synchronous system, Tendermint One-Shot Consensus Algorithm verifies the following property: If $f + 1$ correct processes locked on the same value B during a round r then no correct process can lock during round $r' > r$ on a value $B' \neq B$.*

Proof. We assume that $f + 1$ correct processes are locked on the same value B during the round r, and we denote by X^r the set of those processes. We first prove by induction that no process in X^r will unlock or lock on a new value. Let let $p_i \in X^r$.

- *Initialization*: round $r + 1$. At the beginning of round $r + 1$, all processes in X^r are locked on B. Moreover, we have that $LLR_i = r$, since p_i locks on round r (line 31). Let p_j be the proposer for round $r + 1$. If $LLR_j = r$, it means that p_j is also locked on B, since there cannot be a value $B' \neq B$ such that $\mathsf{is23Maj}(B, \mathit{prevotesReceived}_j^{H,r}) = \mathsf{true}$, for that to happen, at least $n/3$ processes should prevote both B and B' during round r, which means that at least a correct process prevoted two times in the same round, which is not possible, since it is correct, and the protocol does not allow to vote two times in the same round (lines 17 - 29). Three cases can then happen:
  - p_j locked on a value B_j during the round $LLR_j \leq r$. This means that during the round LLR_j $\mathsf{is23Maj}(B_j, \mathit{prevotesReceived}_j^{H,LLR_j}) = \mathsf{true}$ (line 31). p_j the proposer proposes a value B_j along with LLR_j (lines 5 - 9). Since $LLR_j \leq LLR_i = r$, p_i does not unlock and prevotes B for the round $r + 1$, and so are all the other processes in X^r (lines 17 - 21). The only value that can have more than $2n/3$ prevotes is then B. So p_i is still locked on B at the end of $r + 1$.
  - If p_j is not locked, the value it proposes cannot unlock processes in X^r because $-1 = LLR_j < r$, and they will prevote on B (lines 17 - 21). The only value that can have more than $2n/3$ prevotes is then B. So p_i is still locked on B at the end of $r + 1$.
  - p_j locked on a value B_j during the round $LLR_j > r$, p_j the proposer proposes a value B_j along with LLR_j (lines 5 - 9). Since $LLR_j \geq r + 1$, p_i does not unlock and prevotes

B for the round $r+1$, and so are all the other processes in X^r (lines 17 - 21). The only value that can have more than $2n/3$ prevotes is then B. So p_i is still locked on B at the end of $r+1$.

At the end of round $r+1$, all processes in X^r are still locked on B and it may happen that other processes are locked on B for round $r+1$ at the end of the round.

- *Induction*: We assume that for a given $a > 0$, the processes in X^r are still locked on B at each round between r and $r+a$. We now prove that the processes in X^r will still be locked on B at round $r+a+1$.
  Let p_j be the proposer for round $r+a+1$. Since the $f+1$ processes in X^r were locked on B for all the rounds between r and $r+a$, no new value can have more than $2n/3$ of prevotes during one of those rounds, so $\nexists B' \neq B : \mathsf{is23Maj}(B', \mathit{prevotesReceived}_j^{H,r_j}) = \mathsf{true}$ where $r < r_j < r+a+1$. Moreover, if p_j proposed the value B along with a $LLR > r$, since the processes in X^r are already locked on B, they do not unlock and prevote B (lines 17 - 21). The proof then follows as in the *Initialization* case.

Therefore all processes in X^r will stay locked on B at each round after round r. Since $f+1$ processes will stay locked on the value B on rounds $r' > r$, they will only prevote on B (lines 17 - 21) for each new round. Let B' be a value, we have that $\forall r' \geq r$ if $B' : \mathsf{is23Maj}(B', \mathit{prevotesReceived}_j^{H,r'}) = \mathsf{true}$ then $B' = B$. ◀

▶ **Lemma 4** (One-Shot Agreement). *In an eventual synchronous system, Tendermint One-Shot Consensus Algorithm verifies the following property: If there is a correct process that decides a value B, then eventually all the correct processes decide B.*

Proof. Let p_i be a correct process. Without loss of generality, we assume that p_i is the first correct process to decide, and it decides B at round r. If p_i decides B, then $\mathsf{is23Maj}(B, \mathit{precommitsReceived}_i^{H,r}) = \mathsf{true}$ (line 62), since the signature of the messages are unforgeable by hypothesis and $f < n/3$, then p_i delivers more than $n/3$ of those precommits for round r from correct processes, and those correct process are locked on B at round r (line 31). p_i broadcasts all the precommits it delivers (line 56), so eventually all correct processes will deliver those precommits, because of the best effort broadcast guarantees.

We now show that before delivering the precommits from p_i, the other correct processes cannot decide a different value than B. $f < n/3$ by hypothesis, so we have that at least $f+1$ correct processes are locked on B for the round r. By Lemma 3 no correct process can lock on a value different than B. Let $B' \neq B$, since correct processes lock only when they precommit (lines 30 - 32), no correct process will precommit on B' for a round bigger than r, so $\mathsf{is23Maj}(B', \mathit{precommitsReceived}_i^{H,r'}) = \mathsf{false}$ for all $r' \geq r$ since no correct process will precommit on B'. No correct process cannot decide a value $B' \neq B$ (line 62) once p_i decided. Eventually, all the correct processes will deliver the $2n/3$ signed precommits p_i delivered and broadcasted, thanks to the best effort broadcast guarantees and then will decide B. ◀

▶ **Lemma 5.** *In an eventual synchronous system, and under the assumption that during the synchronous period eventually there is a correct proposer p_k such that $|\{p_j : LLR_k \leq LLR_j$ and p_j is correct$\}| < n/3 - f$, Tendermint One-Shot Consensus Algorithm verifies the following property: Eventually a correct process decides.*

Proof. Let r be the round where the communication becomes synchronous and when all the messages broadcasted by correct processes are delivered by the correct processes within their respective step. The round r exists, since the system is eventually synchronous and correct processes increase their time-outs when they did not deliver enough messages (lines 13 - 15,

26 - 29 and 40). If a correct process decides before r, that ends the proof. Otherwise no correct process decided yet. Let p_i be the proposer for the round r. We assume that p_i is correct. Let B be the value such that p_i proposes (B, LLR_i), we have three cases:

- Case 1: No correct process is locked on a value before r. $\forall p_j \in \Pi_\rho$ such that p_j is correct, $LLR_j = -1$.
  Correct processes delivered the proposal (B, LLR_i) before the Prevote step (lines 12, 42 - 44). Since the proposal is valid, then all correct processes will prevote on that value (line 22), and they deliver the others' prevotes and broadcast them before entering the Precommit step (lines 25 - 29 and 48). Then for all correct process p_j, we have $\mathsf{is23Maj}(B, \mathit{precommitsReceived}_j^{H,r}) = \mathsf{true}$. The correct processes will lock on B, precommit on B (lines 30 - 32) and will broadcast all precommits delivered (line 56). Eventually a correct process p_j will have $\mathsf{is23Maj}(B, \mathit{precommitsReceived}_j^{H,r}) = \mathsf{true}$ then p_j will decide (line 62).
- Case 2: Some correct processes are locked and if p_j is a correct process, $LLR_j < LLR_i$. Since $LLR_j < LLR_i$ for all correct processes p_j, then the correct processes that are locked will unlock (line 19) and the proof follows as in the Case 1.
- Case 3: Some correct processes are locked on a value, and there exist a correct process p_j such that $LLR_i \leq LLR_j$.
  - (i) If $|\{p_j : LLR_i \leq LLR_j$ and p_j is correct$\}| < n/3 - f$ (which means that even without the correct processes that are locked in a higher round than the proposer p_i, there are more than $2n/3$ other correct processes unlock or locked in a smaller round than LLR_i), then as in the case 2, a correct process will decide.
  - (ii) If $|\{p_j : LLR_i \leq LLR_j$ and p_j is correct$\}| \geq n/3 - f$, then during the round r, $\nexists B' : \mathsf{is23Maj}(B', \mathit{precommitsReceived}_i^{H,r}) = \mathsf{true}$, in fact correct processes only precommit once in a round (lines 30 - 40). Eventually, thanks to the additional assumption, there exists a round r_1 where the proposer p_k is correct and at round r_1, $|\{p_j : LLR_k \leq LLR_j$ and p_j is correct$\}| < n/3 - f$. The proof then follows as case (3.i).

If p_i is Byzantine and more than $n/3$ correct processes delivered the same message during the proposal step, and the proposal is valid, the situation is like p_i was correct. Otherwise, there are not enough correct processes that delivered the proposal, or if the proposal is not valid, then there will be less than $n/3$ processes that will prevote that value. No value will be committed. Since the proposer is selected in a round robin fashion, a correct process will eventually be the proposer, and a correct process will decide. ◀

▶ **Lemma 6** (One-Shot Termination). *In an eventual synchronous system, and under the assumption that during the synchronous period eventually there is a correct proposer p_k such that $|\{p_j : LLR_k \leq LLR_j$ and p_j is correct$\}| < n/3 - f$, Tendermint One-Shot Consensus Algorithm verifies the following property: Every correct process eventually decides some value.*

Proof. By construction, if a correct process does not deliver a proposal during the proposal step or enough prevotes during the Prevote step, then that process increases its time-outs (lines 13 - 15 and 26 - 29), so eventually, during the synchrony period of the system, all the correct processes will deliver the proposal and the prevotes from correct processes respectively during the Propose and the Prevote step. By Lemma 5, a correct process decides a value, and then by the Lemma 4, every correct process eventually decides. ◀

▶ **Theorem 7.** *In an eventual synchronous system, and under the assumption that during the synchronous period eventually there is a correct proposer p_k such that $|\{p_j : LLR_k \leq LLR_j$ and p_j is correct$\}| < n/3 - f$: Tendermint One-Shot Algorithm implements the One-Shot Consensus.*

```
Function repeatedConsensus(Π_ρ); %Repeated Consensus for the set Π_ρ of processes%

Init:
 (1)  H ← 1 %Height%; B ← ⊥; V ← ⊥ %Set of validators%;
 (2)  commitsReceived_i^H ← ∅; toReward_i^H ← ∅; TimeOutCommit ← Δ_Commit;
------------------------------------------------------------------------
while (true) do
 (3)  B ← ⊥;
 (4)  V ← validatorSet(H); %Application and blockchain dependant%
 (5)  if (p_i ∈ V) then
 (6)      B ← consensus(H, V, toReward_i^{H−1}); %Consensus function for the height H%
 (7)      trigger broadcast ⟨COMMIT, (B, H)_i⟩;
 (8)  else
 (9)      wait until (∃B' : |atLeastOneThird(B', commitsReceived_i^H)|);
 (10)     B ← B';
 (11) endif
 (12) set timerCommit to TimeOutCommit;
 (13) wait until(timerCommit expired);
 (14) trigger decide(B);
 (15) H ← H + 1;
endwhile
------------------------------------------------------------------------
upon event delivery ⟨COMMIT, (B', H')_j⟩:
 (16) if (((B', H')_j ∉ commitsReceived_i^{H'}) ∧ (p_j ∈ validatorSet(H'))) then
 (17)     commitsReceived_i^{H'} ← commitsReceived_i^{H'} ∪ (B', H')_j;
 (18)     toReward_i^{H'} ← toReward_i^{H'} ∪ p_j;
 (19)     trigger broadcast ⟨COMMIT, (B', H')_j⟩;
 (20) endif
```

Figure 4 Tendermint Repeated Consensus algorithm at correct process p_i.

Proof. The proof follows directly from Lemmas 1, 2, 4 and 6. ◀

4.3 Tendermint Repeated Consensus algorithm

For a given height, the set V of validators does not change. Note that each height corresponds to a block. Therefore, in the following we refer this set as the set of validators for a block.

Data structures. The integer H is the height where is called a One-Shot Consensus instance. V is the current set of validators. B is the block to be appended. $commitsReceived_i^H$ is the set containing all the commits p_i delivered for the height H. $toReward_i^H$ is the set containing the validators from which p_i delivered commits for the height H. $TimeOutCommit$ represents the time a process has for collecting commits after an instance of consensus. $TimeOutCommit$ is set to Δ_{Commit}.

Functions.

- validatorSet : $\Pi_\rho \times Height \to 2^{\Pi_\rho}$ is an application dependent and deterministic selection function which gives the set of validators for a given height w.r.t the blockchain history, and $\forall H \in Height, |validatorSet(H)| = n$.
- consensus : $Height \times 2^{\Pi_\rho} \times commitsReceived \to Block$ is the One-Shot Consensus instance presented in 4.2.
- atLeastOneThird : $Block \times commitsReceived \to$ Bool is a predicate which checks if there is at least $n/3$ of commits of the given block in the given set.

Detailed description of the algorithm. In Fig. 4 we describe the algorithm to solve the Repeated Consensus as defined in Section 3. The algorithm proceeds as follows:

- p_i computes the set of validators for the current height;
- If p_i is a validator, then it calls the consensus function solving the consensus for the current height, then broadcasts the decision, and sets B to that decision;

- Otherwise, if p_i is not a validator, it waits for at least $n/3$ commits from the same block and sets B to that block;
- In any case, it sets the timer to *TimeOutCommit* to receive more commits and lets it expire. Then p_i decides B and goes to the next height. We note that this timer is not adjustable, so processes might miss commits from correct processes even during the synchronous period (see fairness study [2] for further details).

Whenever p_i delivers a commit, it broadcasts it (lines 16 - 20). Note that the reward for the height H is given during the height $H+1$, and to a subset of validators who committed the block for H (line 6).

▶ **Theorem 8.** *In an eventual synchronous system, Tendermint Repeated Consensus algorithm implements the Repeated Consensus.*

5 Conclusion & Discussion

The contribution of this paper is the improvement and the formal analysis of the original Tendermint protocol, a PBFT-based repeated consensus protocol where the set of validators is dynamic. Each improvement we introduced is motivated by bugs we discovered in the original protocol. A preliminary version of this paper has been reported in [2]. Very recently a new version of Tendermint has been advertised in [6] by Tendermint foundation without an operational release. The authors argue that their solution works if the two hypothesis below are verified: *Hypothesis 1*: if a correct process receives some message m at time t, all correct processes will receive m before $max(t, global\ stabilization\ time) + \Delta$. Note that this property called by the authors *gossip communication* should be verified even though m has been sent by a Byzantine process. *Hypothesis 2*: there exists eventually a proposer such that its proposed value will be accepted by all the other correct processes. Moreover, the formal and complete correctness proof of this new protocol is still an open issue (several not trivial bugs have been reported recently e.g. [34]).

We are further interested in the *fairness* of Tendermint-core blockchains since without a glimpse of fairness in the way rewards are distributed, these blockchains may collapse. It is common knowledge that in permisionless blockchain systems the main threat is the tragedy of commons that may yield the system to collapse if the rewarding mechanism is not adequate. Ad minimum the rewarding mechanism must be $fair$, i.e. distributing the rewards in proportion to the merit of participants. Our fairness preliminary study, reported in [2], is in line with Francez definition of fairness [18], generally defines the fairness of protocols based on voting committees (e.g. Byzcoin[24], PeerCensus[11], RedBelly [9], SBFT [21] and Hyperledger Fabric [3] etc), by the fairness of their *selection mechanism* and the fairness of their *reward mechanism*. The selection mechanism is in charge of selecting the subset of processes that will participate to the agreement on the next block to be appended to the blockchain, while the reward mechanism defines the way the rewards are distributed among processes that participate in the agreement. Our preliminary analysis of the reward mechanism allowed to establish the following result with respect to the fairness of repeated-consensus blockchains as follows:

There exists a(n) (eventual) fair reward mechanism for repeated-consensus blockchains if and only if the system is (eventual) synchronous (see [2]).

It follows that in our fairness model the original Tendermint protocol is not eventually fair, however with a small twist in the way delays are handled its reward mechanism becomes eventually fair. More details can be found in [2]. Our study opens an interesting future research direction related to the fairness of the selection mechanism in repeated-consensus based blockchains.

References

1. Marcos K Aguilera. A pleasant stroll through the land of infinitely many creatures. *ACM Sigact News*, 35(2):36–59, 2004.
2. Yackolley Amoussou-Guenou, Antonella Del Pozzo, Maria Potop-Butucaru, and Sara Tucci Piergiovanni. Correctness and Fairness of Tendermint-core Blockchains. *CoRR*, abs/1805.08429, 2018.
3. 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 *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 30:1–30:15, 2018.
4. Roberto Baldoni, Marin Bertier, Michel Raynal, and Sara Tucci-Piergiovanni. Looking for a definition of dynamic distributed systems. In *International Conference on Parallel Computing Technologies*, pages 1–14. Springer, 2007.
5. Amotz Bar-Noy, Xiaotie Deng, Juan A. Garay, and Tiko Kameda. Optimal Amortized Distributed Consensus. *Inf. Comput.*, 120(1):93–100, 1995.
6. E. Buchman, J. Kwon, and Z. Milosevic. The latest gossip on BFT consensus. *CoRR*, abs/1807.04938v1, July 2018. URL: `https://arxiv.org/abs/1807.04938v1`.
7. Ethan Buchman. *Tendermint: Byzantine Fault Tolerance in the Age of Blockchains*. Thesis, University of Guelph, June 2016. URL: `https://atrium.lib.uoguelph.ca/xmlui/handle/10214/9769`.
8. Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, November 2002.
9. Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. DBFT: Efficient Byzantine Consensus with a Weak Coordinator and its Application to Consortium Blockchains, 2017.
10. Daian, Rafael Pass, and Elaine Shi. Snow White: Provably Secure Proofs of Stake. *IACR Cryptology ePrint Archive*, 2016:919, 2016.
11. Christian Decker, Jochen Seidel, and Roger Wattenhofer. Bitcoin meets strong consistency. In *Proceedings of the 17th International Conference on Distributed Computing and Networking, Singapore, January 4-7, 2016*, pages 13:1–13:10, 2016.
12. Carole Delporte-Gallet, Stéphane Devismes, Hugues Fauconnier, Franck Petit, and Sam Toueg. With Finite Memory Consensus Is Easier Than Reliable Broadcast. In *Principles of Distributed Systems, 12th International Conference, OPODIS 2008, Luxor, Egypt, December 15-18, 2008. Proceedings*, pages 41–57, 2008.
13. Shlomi Dolev and Sergio Rajsbaum. Stability of long-lived consensus. *J. Comput. Syst. Sci.*, 67(1):26–45, 2003.
14. Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.
15. Cynthia Dwork and Moni Naor. Pricing via Processing or Combatting Junk Mail. In *Advances in Cryptology - CRYPTO '92, 12th Annual International Cryptology Conference, Santa Barbara, California, USA, August 16-20, 1992, Proceedings*, pages 139–147, 1992.
16. Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert van Renesse. Bitcoin-NG: A Scalable Blockchain Protocol. In *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*, 2016.
17. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2), April 1985.
18. Nissim Francez. *Fairness*. Texts and Monographs in Computer Science. Springer, 1986.

19 J. A. Garay, A. Kiayias, and N. Leonardos. The Bitcoin Backbone Protocol: Analysis and Applications. In *Proc. of the EUROCRYPT International Conference*, 2015.

20 Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 51–68, 2017.

21 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 decentralized trust infrastructure for blockchains. *CoRR*, abs/1804.01626, 2018.

22 Maurice Herlihy and Mark Moir. Enhancing Accountability and Trust in Distributed Ledgers. *CoRR*, abs/1606.07490, 2016.

23 Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol. In *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*, pages 357–388, 2017.

24 E. Kokoris-Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford. Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing. In *Proceedings of the 25th USENIX Security Symposium*, 2016.

25 Jae Kwon. Tendermint: Consensus without mining. Technical report, Tendermint, 2014.

26 Jae Kwon and Ethan Buchman. Cosmos: A Network of Distributed Ledgers. `https://cosmos.network/resources/whitepaper` (visited on 2018-05-22).

27 Jae Kwon and Ethan Buchman. Tendermint. `https://tendermint.readthedocs.io/projects/tools/en/v0.19.3/specification.html` (visited on 2018-05-22).

28 Dahlia Malkhi. The BFT Lens: Tendermint. `https://dahliamalkhi.wordpress.com/2018/04/03/tendermint-in-the-lens-of-bft/` (visited on 2018-05-22), April 2018.

29 Francesc D. Muñoz-Escoí and Rubén de Juan-Marín. On synchrony in dynamic distributed systems. *Open Computer Science*, 8(1):154–164, 2018. `doi:10.1515/comp-2018-0014`.

30 S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. `https://bitcoin.org/bitcoin.pdf` (visited on 2018-05-22), 2008.

31 Rafael Pass, Lior Seeman, and Abhi Shelat. Analysis of the Blockchain Protocol in Asynchronous Networks. In *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part II*, pages 643–673, 2017.

32 Rafael Pass and Elaine Shi. The Sleepy Model of Consensus. In *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part II*, pages 380–409, 2017.

33 M. Pease, R. Shostak, and L. Lamport. Reaching Agreement in the Presence of Faults. *Journal of the ACM*, 27(2):228–234, April 1980.

34 Tendermint. Tendermint: correctness issues. `https://github.com/tendermint/spec/issues` (see issues 36-37 visited on 2018-09-05).

35 Tendermint. Tendermint: Tendermint Core (BFT Consensus) in Go. `https://github.com/tendermint/tendermint/blob/e88f74bb9bb9edb9c311f256037fcca217b45ab6/consensus/state.go` (visited on 2018-05-22).

36 G. Wood. Ethereum: A secure decentralised generalised transaction ledger. `http://gavwood.com/Paper.pdf` (visited on 2018-05-22).

Federated Byzantine Quorum Systems

Álvaro García-Pérez
IMDEA Software Institute, Madrid, Spain

Alexey Gotsman
IMDEA Software Institute, Madrid, Spain

Abstract

Some of the recent blockchain proposals, such as Stellar and Ripple, use quorum-like structures typical for Byzantine consensus while allowing for open membership. This is achieved by constructing quorums in a decentralised way: each participant independently chooses whom to trust, and quorums arise from these individual decisions. Unfortunately, the theoretical foundations underlying such blockchains have not been thoroughly investigated. To close this gap, in this paper we study decentralised quorum construction by means of federated Byzantine quorum systems, used by Stellar. We rigorously prove the correctness of basic broadcast abstractions over federated quorum systems and establish their relationship to the classical Byzantine quorum systems. In particular, we prove correctness in the realistic setting where Byzantine nodes may lie about their trust choices. We show that this setting leads to a novel variant of Byzantine quorum systems where different nodes may have different understanding of what constitutes a quorum.

2012 ACM Subject Classification Theory of computation → Distributed computing models

Keywords and phrases Blockchain, Stellar, Byzantine quorum systems

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.17

Related Version An extended version of the paper is available at [8], https://arxiv.org/abs/1811.03642.

Acknowledgements We thank Gregory Chockler for discussions about Byzantine quorum systems, and Ilya Sergey and Maria Schett for discussions about Stellar. This work was supported by an ERC Starting Grant RACCOON.

1 Introduction

Blockchains are distributed databases that maintain an append-only ledger over a set of potentially Byzantine nodes. The nodes use a Byzantine fault-tolerant (BFT) consensus protocol to agree on a total order in which transactions are appended to the ledger. Blockchains usually come in two flavours. *Permissioned* blockchains assume a known set of participants, and are often based on classical BFT consensus protocols, such as PBFT [5]. In these protocols consensus is reached once a *quorum* of participants agrees on the same decision. Quorums can be defined as sets containing enough nodes in the system, e.g., $2f + 1$ out of $3f + 1$, assuming at most f failures. They can also be defined by a more general structure called a *Byzantine quorum system (BQS)* [10] – a pair of a *quorum system* $\mathcal{Q}$ and a *fail-prone system* $\mathcal{B}$, the latter containing the sets of nodes characterising possible failure scenarios [10]. A subclass of *dissemination quorum systems (DQS)* requires $\mathcal{Q}$ and $\mathcal{B}$ to satisfy certain additional axioms, e.g., that any two quorums must intersect in a non-faulty node.

The other kind of blockchains are *permissionless* ones, which allow anyone to participate in consensus, with the membership constantly changing. These blockchains are often based on consensus protocols such as proof-of-work [7, 12], which do not rely on quorums. Despite

© Álvaro García-Pérez and Alexey Gotsman;
licensed under Creative Commons License CC-BY
22nd International Conference on Principles of Distributed Systems (OPODIS 2018).
Editors: Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira; Article No. 17; pp. 17:1–17:16

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

the advantages of open membership, proof-of-work suffers from a number of drawbacks, including high energy consumption and the absence of hard guarantees on when a transaction can be considered successfully appended to the ledger. This has motivated a number of proposals of alternative architectures for permissionless blockchains. In this paper we focus on an intriguing new class of blockchains, such as Stellar [11] and Ripple [13], that use quorum-like structures typical for BFT consensus while allowing open membership. This is achieved by constructing the quorum system in a decentralised way: each protocol participant independently chooses whom to trust, and quorums arise from these individual decisions.

In particular, in Stellar trust assumptions are specified using a *federated Byzantine quorum system (FBQS)*[1], where each participant selects a set of *quorum slices* – sets of nodes each of which would convince the participant to accept the validity of a given statement. Quorums are defined as sets of nodes U such that each node in U has some quorum slice fully within U, so that the nodes in a quorum can potentially reach an agreement. Consensus is then implemented by a fairly intricate protocol whose key component is *federated voting* – a protocol similar to Bracha's protocol for reliable Byzantine broadcast [2, 3]. Importantly, the decentralised nature of Stellar means that its participants operate based on an incomplete and inconsistent information about the trust assumptions of various nodes. First, a node may not have complete knowledge of quorum slices of all other nodes in the system. Second, Byzantine nodes may lie to other nodes about their choices of quorum slices. These features make establishing the correctness of the protocols underlying Stellar nontrivial.

Even though Stellar has been deployed as a functioning blockchain, its theoretical foundations remain shaky. The core Stellar protocols lack rigorous formalisations and proofs of correctness. Furthermore, as observed in [4], the federated quorum structures arising in Stellar are similar to the classical Byzantine quorum systems, where trust assumptions are defined globally for the whole system [10]. But the relationship between these has not been investigated, and this is a barrier to transferring ideas between the classical and the federated settings. In this paper we take the first step towards closing these gaps and perform a rigorous theoretical study of the basic concepts underlying the Stellar blockchain.

In more detail, the closest protocol to Stellar's core federated voting protocol in the world of classical Byzantine quorum systems is *Bracha broadcast* [2]. This implements *reliable Byzantine broadcast* [3] in a system of $3f + 1$ processes where at most f processes can fail and any $2f + 1$ processes constitute a quorum. The protocol allows a node to send a message to a group of receivers; the sender may be faulty and can thus send different messages to different receivers. The protocol guarantees, among other properties, that: *(i)* correct receivers cannot deliver different messages; and *(ii)* if a correct receiver delivers a message, then eventually all correct receivers will deliver the same message [3]. Bracha's protocol can also be executed over an arbitrary dissemination quorum system (which we prove in §2).

We investigate Stellar's federated voting through the prism of the reliable Byzantine broadcast abstraction. We first consider an idealised setting (used in the Stellar whitepaper [11]) where nodes have consistent information about the quorum slices of nodes they interact with, i.e., faulty nodes do not equivocate about their quorum slices, but can otherwise behave arbitrarily. Thus, all nodes act according to the same FBQS. Our first contribution is to show that, in this setting, a federated Byzantine quorum system used in Stellar induces a classical Byzantine quorum system $(\mathcal{Q}, \mathcal{B})$, so that protocols such as Bracha broadcast

[1] In the Stellar whitepaper these are called *federated Byzantine agreement systems* [11]. The name used in this paper is chosen to be in line with common terminology [10] and to emphasise that these systems can be used for purposes other than solving consensus.

can also be run out of the box in a federated setting (§3). However, Bracha broadcast requires a node to know the fail-prone system $\mathcal{B}$, and computing it from an FBQS would require the node to know the quorum slices of *all* the nodes in the system. This is infeasible in a system with open membership. Stellar circumvents this problem by allowing a node to take decisions based solely on the quorum slices of the nodes it interacts with directly. Our next contribution is to prove that the resulting *Stellar broadcast* implements what we call *weakly reliable Byzantine broadcast* (§4). This guarantees the safety property (i) of the usual reliable Byzantine broadcast and the liveness property (ii) where only certain *intact nodes* are guaranteed to deliver the message. In a given protocol execution, the set of intact nodes is the largest set that excludes all faulty nodes and satisfies certain basic properties allowing correct protocol operation among these nodes, e.g., that every two quorums intersect. This result strengthens the correctness theorems in the Stellar whitepaper [11], which only guarantee safety to intact nodes (instead of all correct ones) and guarantee liveness only under an assumption that an intact node delivers a message (rather than only correct one). We also show that a variation of the Stellar protocol actually implements the usual reliable Byzantine broadcast and prove that this variation is observationally equivalent to Bracha broadcast over the corresponding DQS. On a conceptual level, our correctness results show that some of the ad hoc protocols proposed for Stellar actually implement a variant of a well-known broadcast abstraction.

We next consider a more realistic setting where faulty nodes may lie about their choices of quorum slices (§5), which has not been covered by the existing correctness theorems for Stellar [11]. In this setting, different nodes may have different understanding of what the quorum slices of a given faulty node are. We capture this by the notion of a *subjective FBQS*, where each node has its own, possibly different, copy of an FBQS such that the different FBQSes agree on the quorum slices of correct nodes. This also leads different nodes to disagree on what constitutes a quorum. More precisely, a subjective FBQS induces a *subjective quorum system*, where each node has its own copy of a quorum system such that the projections of different quorum systems to correct nodes coincide. Our next result is to show that the Stellar broadcast correctly implements weakly reliable Byzantine broadcast even when each node acts according to its subjective view on the trust choices in the system. This result is required to get confidence in the correctness of the Stellar blockchain, since in its intended deployment faulty nodes may lie about everything, including their trust choices.

Finally, we generalise the theory of classical Byzantine quorum systems to allow for nodes to have inconsistent information about quorums, regardless of the origin of this inconsistency (§6); this may be useful in the future for dealing with approaches to decentralising trust different from Stellar. To this end, we introduce the notion of a *subjective dissemination quorum system*, where each node has its own Byzantine quorum system such that the different systems satisfy a variation on the DQS axioms. For example, we require that any two quorums, *even when coming from the systems of different nodes*, have to intersect in a non-faulty node. We show that Bracha broadcast over a subjective DQS implements reliable Byzantine broadcast. Furthermore, we show that a subjective FBQS induces a subjective DQS, on top of which one can execute Bracha broadcast.

Our results open the door to a deeper theoretical study of the novel quorum structures arising in a federated setting and to proving the correctness of the whole Stellar consensus protocol. We also hope that the connections we have established between classical broadcast abstractions and the novel Stellar protocols will enable transferring ideas from the existing optimised protocols for BFT consensus [9, 6] into the federated setting.

Due to space constraints, proofs are deferred to an extended version of the paper [8].

2 Byzantine Quorum Systems and Bracha Broadcast

System model. In this paper we consider a system consisting of a set of *client nodes* $\mathbf{C}$ and a disjoint set of *server nodes* $\mathbf{V}$. We assume a Byzantine failure model: some nodes may be *faulty*, in which case they can deviate arbitrarily from their specification. All other nodes are called *correct*. We assume that any two nodes (clients or servers) can communicate over an asynchronous point-to-point channel. If both endpoints of the channel are correct, then this channel is both authenticated and reliable: a correct node receives a message from another correct node if and only if the latter node sent it.

Byzantine quorum systems. A *quorum system* is a non-empty set $\mathcal{Q}$ of subsets of some universe of nodes, such that

- every quorum in $\mathcal{Q}$ is non-empty;
- any two quorums in $\mathcal{Q}$ intersect: $\forall U_1, U_2 \in \mathcal{Q}.\, U_1 \cap U_2 \neq \emptyset$;
- $\mathcal{Q}$ is closed under union: $\forall U_1, U_2 \in \mathcal{Q}.\, U_1 \cup U_2 \in \mathcal{Q}$.

In this paper we let the universe of nodes coincide with the set of servers $\mathbf{V}$, so that $\mathcal{Q} \subseteq 2^{\mathbf{V}}$. A *fail-prone system* $\mathcal{B}$ is a non-empty set of subsets of $\mathbf{V}$ such that none of its elements is contained in another. A fail-prone system characterises the failure scenarios that can occur: in any execution of the system, some $B \in \mathcal{B}$ is meant to contain all the faulty servers. A *Byzantine quorum system (BQS)* is a pair $(\mathcal{Q}, \mathcal{B})$ of a quorum system and a fail-prone system. We sometimes use the adjective ‘classical’ to distinguish such quorum systems from federated ones that we define in §3.

In this paper we focus on a particular class of BQSes that satisfy additional properties. Namely, a *dissemination quorum system (DQS)* is a BQS with the following properties [10]:

- *D-consistency:* $\forall U_1, U_2 \in \mathcal{Q}.\, \forall B \in \mathcal{B}.\, U_1 \cap U_2 \not\subseteq B$; and
- *D-availability:* $\forall B \in \mathcal{B}.\, \exists U \in \mathcal{Q}.\, U \cap B = \emptyset$.

Informally, D-consistency ensures that any two quorums must intersect in a correct server. D-availability ensures the existence of a quorum containing only correct servers.

▶ **Example 1.** Consider a universe $\mathbf{V}$ with $3f+1$ servers, a quorum system $\mathcal{Q}$ that contains every set with at least $2f+1$ servers, and a fail-prone system $\mathcal{B}$ that contains every set with exactly f servers. Then $(\mathcal{Q}, \mathcal{B})$ is a DQS. Indeed, D-consistency holds because any two quorums have at least $f+1$ servers in common, at least one of which must be correct. D-availability holds because, for each $B \in \mathcal{B}$, the set $\mathbf{V} \setminus B$ has $2f+1$ servers and is thus a quorum in $\mathcal{Q}$.

▶ **Example 2.** Consider the universe $\mathbf{V} = \{1, 2, 3, 4\}$, the quorum system

$$\mathcal{Q} = \{\{1,2\},\ \{1,2,3\},\ \{1,3,4\},\ \{1,2,3,4\}\}$$

and the fail-prone system $\mathcal{B} = \{\{2\}, \{3,4\}\}$. Then $(\mathcal{Q}, \mathcal{B})$ is a DQS. D-consistency holds because all quorums in $\mathcal{Q}$ intersect at 1, which does not belong to any element of $\mathcal{B}$. D-availability holds because both $\{1,3,4\}$ and $\{1,2\}$ are quorums.

Bracha broadcast over DQS. We next consider the abstraction of *reliable Byzantine broadcast* and show that a generalisation of Bracha broadcast [2] implements it when executed over an arbitrary dissemination quorum system. This protocol is the closest one from the world of classical BQSes that matches the broadcast protocol used in Stellar, and we relate the two in the following sections.

Algorithm 1: Bracha broadcast over an arbitrary DQS $(\mathcal{Q}, \mathcal{B})$.

```
1  process client(c ∈ C)
2      broadcast(a)
3          send BCAST(a) to every v ∈ V;
4  process server(v ∈ V)
5      echoed, ready, delivered ← ff ∈ Bool;
6      when received BCAST(a) from c and echoed = ff
7          echoed ← tt;
8          send ECHO(a) to every v′ ∈ V;
9      when received ECHO(a) from every u ∈ U for some U ∈ Q and ready = ff
10         ready ← tt;
11         send READY(a) to every v′ ∈ V;
12     when received READY(a) from every u ∈ B for some B ∈ 2^V \ {∅} such
        that ∀B′ ∈ B. B ⊈ B′ and ready = ff
13         ready ← tt;
14         send READY(a) to every v′ ∈ V;
15     when received READY(a) from every u ∈ U for some U ∈ Q and
        delivered = ff
16         delivered ← tt;
17         deliver(a);
```

The broadcast allows a distinguished client to broadcast a value a to a group of servers, an event which we denote $\mathsf{broadcast}(a)$. We denote the event of a server delivering a value a as $\mathsf{deliver}(a)$. Both the sender and some of the receivers can be Byzantine. *Reliable Byzantine broadcast* is defined by the following properties [3]:

- *Validity:* If the sender client is correct and broadcasts a value a, then every correct server eventually delivers a.
- *No duplication:* Every correct server delivers at most one value.
- *Integrity:* If some correct server delivers a value a and the sender client is correct, then a was previously broadcast by the sender.
- *Consistency:* If some correct server delivers a value a and another correct server delivers a value a', then $a = a'$.
- *Totality:* If a correct server delivers a value, then every correct server eventually delivers a value.

The Validity property ensures that the broadcast operates as expected when the sender is correct. When the sender is faulty, the above properties guarantee that servers may not deliver contradictory values, even if the sender sends different values to different servers. In this case they may also not deliver any value at all, but the liveness property of Totality ensures that all correct servers behave the same in this respect.

In Algorithm 1 we present an implementation of reliable Byzantine broadcast over a DQS $(\mathcal{Q}, \mathcal{B})$. This generalises *Bracha's protocol* [2] for reliable Byzantine broadcast, which works over the cardinality-based DQS from Example 1. In order to broadcast a value a, the client sends a $\texttt{BCAST}(a)$ message to every server (line 2). Servers process client messages in several phases, with progress denoted by several Boolean flags (line 5).

Since a faulty client may send contradictory messages to different servers, to ensure Consistency the servers first cross-check the client's proposals. A server that receives a message $\texttt{BCAST}(a)$ for the first time sends an $\texttt{ECHO}(a)$ message to all servers (including itself, for uniformity; line 6). When the server receives a quorum of $\texttt{ECHO}(a)$ messages, it sends a $\texttt{READY}(a)$ message to every server, signalling its willingness to deliver the value a (line 9). Note that correct servers cannot send READY messages in this way with two different values: this would require two quorums of ECHO messages with different values, but by D-consistency these quorums would intersect in a correct server, which cannot send contradictory ECHO messages. When a server receives a quorum of $\texttt{READY}(a)$ messages, it delivers a (line 15).

The exchange of READY messages is necessary to ensure Totality: it guarantees that, if a correct server delivers a value, other correct servers have enough information to also deliver the value. This relies on an additional rule in line 12, allowing a server to send a $\texttt{READY}(a)$ message even if it previously echoed a different value: this is done if the server receives $\texttt{READY}(a)$ from each member of a set B that is not a subset of any element of $\mathcal{B}$. In this case at least one of the servers in B must be correct, and hence, we can trust that the value it proposes has been cross-checked for consistency by a quorum.

▶ **Theorem 3.** *Let $(\mathcal{Q}, \mathcal{B})$ be a DQS. Then the protocol in Algorithm 1 satisfies the specification of reliable Byzantine where all faulty servers belong to some element of $\mathcal{B}$.*

The next example shows that removing the handler at line 12 of Algorithm 1 would undermine Totality.

▶ **Example 4.** Consider the DQS $(\mathcal{Q}, \mathcal{B})$ from Example 1, where quorums and fail-prone sets are defined by their cardinalities, and let $f = 1$ with a total number of 4 servers. As before, assume servers 1, 2 and 4 are correct, and 3 is faulty. Let the sender be faulty and send contradictory $\texttt{BCAST}(a)$ and $\texttt{BCAST}(a')$ messages to the sets $\{1, 2\}$ and $\{4\}$, respectively. Imagine faulty 3 sends $\texttt{ECHO}(a)$ to 1 and 2 and never sends anything to 4. Then both 1 and 2 will send $\texttt{READY}(a)$ to every server thanks to the conditions in line 9. If now 3 sends $\texttt{READY}(a)$ to 1 and 2, then both will deliver value a thanks to the conditions in line 15. However, correct 4 cannot send $\texttt{READY}(a)$ and thus cannot deliver a, because it never receives $\texttt{ECHO}(a)$ from a quorum. But correct 1 and 2 did deliver a. By enabling the handler at line 12, 4 would send $\texttt{READY}(a)$ to every server after receiving $\texttt{READY}(a)$ from 1 and 2, because any set of two servers is not contained in any of the elements of $\mathcal{B}$, which are singletons. After receiving $\texttt{READY}(a)$ from 1, 2 and 4, server 4 will also deliver a thanks to the conditions in line 15.

The next example demonstrates that delivering values right after receiving a quorum of ECHO messages would also undermine Totality, thus motivating the need for READY messages.

▶ **Example 5.** Consider the DQS $(\mathcal{Q}, \mathcal{B})$ and the set of faulty servers from Example 4 and assume that the sender is faulty and sends contradictory $\texttt{BCAST}(a)$ and $\texttt{BCAST}(a')$ messages to the sets $\{1\}$ and $\{2, 4\}$, respectively. Imagine 3 sends $\texttt{ECHO}(a')$ to 2. Then 2 receives $\texttt{ECHO}(a')$ from the quorum $\{2, 3, 4\}$. Imagine now that the faulty server 3 stops sending messages. If servers delivered values after receiving a quorum of ECHO messages, then correct server 2 would deliver a', but correct servers 1 and 4 would never deliver any value.

3 Federated Byzantine Quorum Systems

FBQS definition. We now consider *federated Byzantine quorum systems (FBQS)*, which are used in the Stellar blockchain to construct quorums from the trust choices of individual participants [11]. An FBQS is a function $\mathcal{S} : \mathbf{V} \to 2^{2^{\mathbf{V}}} \setminus \{\emptyset\}$ that specifies the set of *quorum*

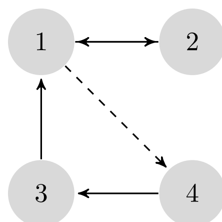
slices for each server, ranged over by q. We require that a server belong to all of its own quorum slices: $\forall v \in \mathbf{V}.\, \forall q \in \mathcal{S}(v).\, v \in q$. Quorum slices reflect the trust choices of each server: a server can be convinced to accept the validity of a statement by any of its slices.

An FBQS specifies quorum slices for all servers. However, to participate in a protocol such as reliable broadcast or consensus, a server may not need to know the whole FBQS, which is infeasible in a system with open membership. Instead, it may be sufficient for the server to know only the quorum slices for the servers it interacts with, which it can receive directly from these servers. A complication is that faulty servers may lie about their quorum slices and, in particular, give contradictory information to their peers. In this section we consider an idealised setting (used in the Stellar whitepaper [11]) that assumes this does not happen: faulty servers do not equivocate about their quorum slices, even though they can otherwise behave arbitrarily. Hence, every server knows a part of the same FBQS $\mathcal{S}$. We lift this assumption in §5.

A non-empty set of servers $U \subseteq \mathbf{V}$ is a *quorum* in an FBQS $\mathcal{S}$ if U contains a slice for each member: $\forall v \in U.\, \exists q \in \mathcal{S}(v).\, q \subseteq U$. Note that a server can check whether U is a quorum based solely on the quorum slices of its participants. Hence, a server can discover quorums without knowing a complete FBQS by probing different servers until it finds a set forming a quorum. Alternatively, (signed) information about quorum slices of a given server can be transitively propagated via chains of trust dependencies, with each server receiving information from the servers in its slices. We say that an FBQS enjoys *quorum intersection* when any two of its quorums intersect. An FBQS $\mathcal{S}$ with quorum intersection induces a quorum system $\mathcal{Q}$ consisting of its quorums.

▶ **Example 6.** Consider an FBQS with $3f+1$ servers where every server has a slice for each set of $2f+1$ servers. This FBQS induces the quorum system $\mathcal{Q}$ from from Example 1: any set of $2f+1$ or more servers is a quorum.

▶ **Example 7.** Consider the following FBQS, where each server has only one slice determined by outgoing arrows, except server 1, which has two slices respectively determined by the solid and the dashed arrows (we omit self-loops in the diagram to avoid clutter). This FBQS induces the quorum system $\mathcal{Q}$ from Example 2.



$$\begin{aligned}\mathcal{S}(1) &= \{\{1,2\},\{1,4\}\}\\ \mathcal{S}(2) &= \{\{1,2\}\}\\ \mathcal{S}(3) &= \{\{1,3\}\}\\ \mathcal{S}(4) &= \{\{3,4\}\}\end{aligned}$$

Intact servers. The correctness of protocols over FBQSes is stated using the following notion of *intact* servers [11]. Informally, the set of intact servers is the biggest one that excludes all faulty servers and still yields a quorum system that satisfies certain properties allowing correct protocol operation within this set. Formally, consider an FBQS $\mathcal{S}$ and a set of servers I. We define the *projection* $\mathcal{S}|_I$ of $\mathcal{S}$ to I as the FBQS over universe I defined as follows: $\mathcal{S}|_I(v) = \{q \cap I \mid q \in \mathcal{S}(v)\}$. Given a set of faulty servers $\mathbf{V}_{\text{bad}}$, the set of *intact* servers $\mathbf{V}_{\text{int}}$ is the biggest set such that

- all servers in $\mathbf{V}_{\text{int}}$ are correct: $\mathbf{V}_{\text{int}} \cap \mathbf{V}_{\text{bad}} = \emptyset$;
- if $\mathbf{V}_{\text{int}}$ is non-empty, then it is a quorum in $\mathcal{S}$;
- $\mathcal{S}|_{\mathbf{V}_{\text{int}}}$ enjoys quorum intersection.

We call the servers in $\mathbf{V} \setminus \mathbf{V}_{\text{int}}$ *befouled*. Informally, even though befouled servers may be correct, they "trust wrong guys" and thus protocols we consider may not guarantee all correctness properties for them.

▶ **Proposition 8.** *The set of intact servers in an FBQS with quorum intersection is well-defined.*

For the FBQS from Example 6, which corresponds to a cardinality-based DQS, the notions of correct and intact coincide. However, this is not the case for the FBQS $\mathcal{S}$ from Example 7. Namely, assume servers 1, 2 and 4 are correct and server 3 is faulty. The set $\{1, 2\}$ is a quorum in $\mathcal{S}$, and $\mathcal{S}|_{\{1,2\}}$ satisfies quorum intersection. Furthermore, adding the correct server 4 to the set $\{1, 2\}$ does not result in a quorum in $\mathcal{S}$. Hence, servers 1 and 2 are intact and server 4 is correct, but befouled. Intact servers satisfy the following useful property [11], which is similar to the D-consistency property of DQSes.

▶ **Proposition 9.** *Consider an FBQS $\mathcal{S}$ with quorum intersection. Fix a set of faulty servers and assume that there exists at least one intact server in $\mathcal{S}$. Then the intersection of every two quorums in $\mathcal{S}$ contains some intact server.*

From federated to classical quorum systems. We now show that every FBQS induces a corresponding DQS. An FBQS does not explicitly specify the possible failure scenarios, since in a setting with open membership one cannot easily predict them. Hence, to construct a fail-prone system for an FBQS, we consider all possible failures that do not bring the whole system down, i.e., leave at least some servers intact.

▶ **Theorem 10.** *Consider an FBQS $\mathcal{S}$ with quorum intersection. Let $\mathcal{Q}$ be the quorum system it induces and let $\mathcal{B}$ consist of the maximal sets B such that the failure of B leaves the set of intact servers in $\mathcal{S}$ non-empty. Then $(\mathcal{Q}, \mathcal{B})$ is a DQS.*

We say that FBQS $\mathcal{S}$ *induces* the DQS $(\mathcal{Q}, \mathcal{B})$ from the theorem. For example, the FBQS $\mathcal{S}$ from Example 6 induces the DQS from Example 1. Similarly, the FBQS $\mathcal{S}$ from Example 7 induces the DQS from Example 2. Indeed, the failure of server 2 leaves 1, 3 and 4 intact, and the failure of 3 and 4 leaves 1 and 2 intact. Extending either of these failure sets leaves no servers intact. The correspondence stated by Theorem 10 allows us to run any off-the-shelf protocol over a DQS on top of an FBQS with at least one intact server. In particular, the following corollary of Theorems 3 and 10 shows that this is the case for Bracha broadcast.

▶ **Corollary 11.** *Let $\mathcal{S}$ be an FBQS with quorum intersection and $(\mathcal{Q}, \mathcal{B})$ be the DQS it induces. Then the protocol in Algorithm 1 satisfies the specification of reliable Byzantine broadcast in executions where at least one server in $\mathcal{S}$ is intact.*

Note that, unlike in Theorem 3, where the assumption on the allowed failure scenarios was stated using a given $\mathcal{B}$, here this is done by requiring the existence of at least one intact server. For example, consider the FBQS from Example 6. The existence of an intact server in this FBQS is equivalent to requiring that no more than f servers fail, which corresponds to the standard requirement for the cardinality-based DQS (Example 1).

4 Stellar Broadcast and its Correctness

Stellar broadcast. Despite Corollary 11, using Bracha broadcast in a federated setting is problematic. While servers can discover quorums in $\mathcal{Q}$ without knowing the whole FBQS, Bracha broadcast also requires participants to know the fail-prone system $\mathcal{B}$ (line 12).

Computing the $\mathcal{B}$ induced by $\mathcal{S}$ requires knowing the quorum slices of all participants, which is infeasible in a system with open membership. Stellar solves this problem by replacing the condition in line 12 by one that a server can check locally. We now present and prove correct the corresponding broadcast protocol, which is a reformulation of Stellar's *federated voting* protocol; we call the resulting protocol the *Stellar broadcast*.

Given an FBQS $\mathcal{S}$ and a server v, a set of servers B is called v-*blocking* if B overlaps each quorum slice in $\mathcal{S}(v)$. The Stellar broadcast is obtained from Bracha broadcast in Algorithm 1 by making the following changes to three lines:

9: **when received** ECHO(a) **from each** $u \in U$ **for some** $U \in \mathcal{Q}$ ***such that*** $v \in U$
and ready $= \mathit{ff}$;
12: **when received** READY(a) **from every** $u \in B$ ***for some*** v***-blocking*** $B \in 2^{\mathbf{V}} \setminus \{\emptyset\}$
and ready $= \mathit{ff}$;
15: **when received** READY(a) **from every** $u \in U$ **for some** $U \in \mathcal{Q}$
such that $v \in U$ **and** delivered $= \mathit{ff}$,

where v is the server executing the code. The new checks at lines 9 and 15 require a server to only accept information from quorums it belongs to. The new check at line 12 depends only on the slices of v and can thus be performed locally. This allows v to operate even when knowing only the part of the FBQS corresponding to the servers it directly interacts with. The role of the new check in line 12 resembles that of the old check in Algorithm 1. The old check guaranteed that the server v has received at least one READY(a) message from a *correct* server. Due to the following proposition, the new check guarantees that, *if v is intact*, then it has received at least one READY(a) message from an *intact* server.

▶ **Proposition 12.** *Assume an FBQS $\mathcal{S}$ with quorum intersection. For a given set of faulty servers, the set $\mathbf{V}_{\mathrm{bef}}$ of befouled servers is not v-blocking for any intact v.*

Proof. Consider an intact server v. The set of all intact servers $\mathbf{V}_{\mathrm{int}} = \mathbf{V} \setminus \mathbf{V}_{\mathrm{bef}}$ is a quorum in $\mathcal{S}$. Then there exists a slice $q \in \mathcal{S}(v)$ such that $q \subseteq \mathbf{V}_{\mathrm{int}}$. Hence, $\mathbf{V}_{\mathrm{bef}}$ cannot overlap all of v's slices, as required. ◀

As it happens, the Stellar broadcast does not implement reliable Byzantine broadcast, but only its weaker version, which we call *weakly reliable Byzantine broadcast*. This weakens the properties of Validity and Totality (§2) by guaranteeing them only to intact servers:

- *Validity for intact servers:* If the sender client is correct and broadcasts a value a, then every intact server eventually delivers a.
- *Totality for intact servers:* If a correct server delivers a value, then every intact server eventually delivers a value.

As we noted above, for the cardinality-based quorum system from Example 1 the notions of intact and correct coincide. Hence, for this quorum system the weak and the classical broadcast abstractions coincide as well.

▶ **Theorem 13.** *Let $\mathcal{S}$ be an FBQS with quorum intersection. The protocol in Algorithm 1 over $\mathcal{S}$ with the changed lines 9, 12 and 15 satisfies the specification of weakly reliable Byzantine broadcast in executions where at least one server in $\mathcal{S}$ is intact.*

As the next example illustrates, the Stellar broadcast does not satisfy the Totality property of reliable Byzantine broadcast.

▶ **Example 14.** Consider the FBQS $\mathcal{S}$ from Example 7. Assume the sender is faulty and it sends contradictory BCAST(a) and BCAST(a') messages to the sets $\{1, 2\}$ and $\{4\}$ respectively. Servers 1 and 2 will receive ECHO(a) from the quorum $\{1, 2\}$, and by the conditions in the changed lines 9 and 15, they will then send READY(a) to every server and deliver value a. However, 4 will not be able to deliver a, because it will receive READY(a) from the set $\{1, 2\}$, which is a quorum to which 4 does not belong.

The condition in the Stellar broadcast that a server only accepts information from quorums it belongs to (lines 9 and 15) is not strictly required to allow a server to operate without the knowledge of the whole system: a server can ensure that a set of servers is a quorum based on the quorum slices of its participants, regardless of whether the server is one of them. If we drop the above condition, then the protocol implements the usual reliable Byzantine broadcast.

▶ **Theorem 15.** *Let $\mathcal{S}$ be an FBQS with quorum intersection. The protocol in Algorithm 1 over $\mathcal{S}$ with the changed line 12 satisfies the specification of reliable Byzantine broadcast in executions where at least one server in $\mathcal{S}$ is intact.*

The above theorems yield several novel insights with respect to the existing results about Stellar. First, they show that some of the ad hoc protocols proposed for Stellar actually implement a variant of a well-known broadcast abstraction. Second, the theorems strengthen the existing correctness theorems for Stellar's federated voting [11]. Namely, we guarantee Consistency for all correct servers, whereas the Consistency property proved for federated voting [11, Theorem 9] applies only to a subset of correct servers I such that $\mathcal{S}|_I$ enjoys quorum intersection. The difference is significant in practice, since a server cannot easily find out if it belongs to such a subset. The Totality property proved for federated voting [11, Theorem 11] requires an intact server to deliver a value, whereas we show that a correct server delivering a value is enough for all intact servers to also deliver the value. Our theorems also show that, assuming the existence of at least one intact server, the guarantees provided by the protocol can be strengthened if a server accepts information from quorums it does not participate in. In §5 we strengthen our correctness statements even further by considering the case when faulty servers may equivocate about their quorum slices.

Proving correctness. We give the key steps in proving Theorem 13, which we revisit in §5 to show that the protocol also works when faulty servers equivocate about their quorum slices. The following lemma is used to establish Consistency.

▶ **Lemma 16.** *Assume an FBQS $\mathcal{S}$ with quorum intersection. If intact servers in an execution of Stellar broadcast over $\mathcal{S}$ send messages* READY(a) *and* READY(a')*, then $a = a'$.*

Proof sketch. To prove this lemma, using Proposition 12 we establish that, if intact servers send messages READY(a) and READY(a'), then some (possibly different) intact servers receive quorums of messages ECHO(a) and ECHO(a'), respectively. By Proposition 9 these quorums have to intersect in an intact server, which is also correct and thus cannot send contradictory ECHO messages. ◀

Proof sketch for Consistency. Assume that correct servers deliver values a and a'. Then by the condition in the changed line 15, the corresponding servers received a quorum of READY(a) and READY(a') messages, respectively. Since at least one server in $\mathcal{S}$ is intact, the set of intact servers forms a quorum. Then, since $\mathcal{S}$ has quorum intersection, each of the quorums that sent the READY(a) and READY(a') messages must contain at least one intact server. But then by Lemma 16 we have $a = a'$, as required. ◀

The next lemma is used to prove Totality.

▶ **Lemma 17.** *Let $\mathcal{S}$ be an FBQS with quorum intersection. Consider an execution of the Stellar broadcast with the set of intact servers in $\mathcal{S}$ being $\mathbf{V}_{\text{int}} \neq \emptyset$. Assume that $\mathbf{V}_{\text{int}} = V^+ \uplus V^-$ and for some quorum U we have $U \cap \mathbf{V}_{\text{int}} \subseteq V^+$. Then either $V^- = \emptyset$ or there exists some server $v \in V^-$ such that V^+ is v-blocking.*

Proof sketch for Totality. Delivering a value a at an intact server requires this server to receive $\texttt{READY}(a)$ messages from a quorum U. Since the set of intact servers forms a quorum and any two quorums intersect, the quorum U must contain some intact servers. Using the above lemma, we can then show that, as the $\texttt{READY}(a)$ messages from these intact servers propagate through the system, more and more intact servers will execute the handler at line 12 and send $\texttt{READY}(a)$ messages themselves. When applying the lemma, the set V^+ is the set of intact servers that have sent $\texttt{READY}(a)$ and V^- the set of intact servers that have not done so yet. The lemma ensures that in the end all intact servers will send $\texttt{READY}(a)$ and will be able to deliver a. ◀

Observational equivalence. We now establish a tighter correspondence between the Stellar broadcast over an FBQS and Bracha broadcast over the corresponding BQS: we show that Bracha broadcast is observationally equivalent to the version of the Stellar broadcast with the changed line 12 only. Informally, this means that any externally observable behaviour of one of the protocols can also be produced by the other. Note that this relationship does not simply follow from the fact that the two broadcasts implement the same specification, because the specification is non-deterministic: when the sender is faulty and sends contradictory values, it allows receivers to pick one value to deliver, or not to deliver any value at all. The equivalence thus shows that, whenever servers in one of the protocols decide to deliver a value, there is an execution of the other protocol (with some behaviour of faulty servers) that also delivers the same value.

We formulate the equivalence using the following notion. A *history* H is a set of events in an execution of a broadcast protocol that includes the reception of the first `BCAST` message received by each server and the triggering of the deliver primitive by each server. A history captures the initial communication between the clients and the system and the outcome of the protocol's execution that is observable by applications running at the servers.

▶ **Theorem 18.** *Let $\mathcal{S}$ be an FBQS with quorum intersection. Fix the sets of faulty and correct servers such that at least one intact server exists in $\mathcal{S}$, and let $(\mathcal{Q}, \mathcal{B})$ be the DQS that $\mathcal{S}$ induces.*

(i) *For every execution of Bracha broadcast over $(\mathcal{Q}, \mathcal{B})$ with a history H, there exists an execution of the Stellar broadcast over $\mathcal{S}$ with the changed line 12 that also entails H.*

(ii) *For every execution of the Stellar broadcast over $\mathcal{S}$ with the changed line 12 that has a history H, there exists an execution of Bracha broadcast over $(\mathcal{Q}, \mathcal{B})$ with a history H.*

5 Lying about Trust Choices

So far we have assumed that faulty servers do not equivocate about their quorum slices, so that all servers share the same FBQS $\mathcal{S}$. This is unrealistic, since in practice servers find out about quorum slices of their peers from the peers themselves, and faulty peers may lie. This may lead different servers to have inconsistent information about the quorum slices of a given faulty server. To capture this, we generalise the notion of an FBQS to allow different servers to have different views on quorum slices. Let us fix the sets of correct and faulty

servers, $\mathbf{V}_{\text{ok}}$ and $\mathbf{V}_{\text{bad}}$. We say that an indexed family $\{\mathcal{S}_v\}_{v\in\mathbf{V}_{\text{ok}}}$ is a *subjective FBQS* if each $\mathcal{S}_v$ is an FBQS and the different FBQSes agree on the quorum slices of correct servers: $\forall v_1, v_2, v \in \mathbf{V}_{\text{ok}}.\, \mathcal{S}_{v_1}(v) = \mathcal{S}_{v_2}(v)$. A subjective FBQS reflects each correct server's view on the choices of trust of other servers. We thus sometimes call $\mathcal{S}_v$ the *view* of the server v. We say that q is u's slice *known by* v if $q \in \mathcal{S}_v(u)$. Note that a subjective FBQS does not specify the views of faulty servers, as they are immaterial.

The fact that different servers may have inconsistent information about quorum slices means that the same holds for quorums. Given a subjective FBQS $\{\mathcal{S}_v\}_{v\in\mathbf{V}_{\text{ok}}}$, we say that U is a *quorum known by* v when U is a quorum in FBQS $\mathcal{S}_v$. As before, a server can discover (subjective) quorums by asking its peers for their slices until it finds a set forming a quorum. A subjective FBQS $\{\mathcal{S}_v\}_{v\in\mathbf{V}_{\text{ok}}}$ satisfies *quorum intersection* if $\mathcal{S}_v$ satisfies quorum intersection for every v that is correct. A subjective FBQS $\{\mathcal{S}_v\}_{v\in\mathbf{V}_{\text{ok}}}$ with quorum intersection induces a *subjective quorum system* $\{\mathcal{Q}_v\}_{v\in\mathbf{V}_{\text{ok}}}$, consisting of the quorums known by each correct server.

▶ **Example 19.** Let $\mathbf{V} = \{1, 2, 3, 4\}$ and assume that 3 is faulty and the rest are correct. Consider a subjective FBQS defined by the following diagrams:

Each server's slices are determined by the outgoing arrows (we omit self-loops). In the case of server 1 we have the two slices determined respectively by the solid and the dashed arrows. Hence, we have

$$\mathcal{S}_1(1) = \mathcal{S}_2(1) = \mathcal{S}_4(1) = \{\{1,2\},\{1,4\}\}; \qquad \mathcal{S}_1(2) = \mathcal{S}_2(2) = \mathcal{S}_4(2) = \{\{1,2\}\};$$
$$\mathcal{S}_1(4) = \mathcal{S}_2(4) = \mathcal{S}_4(4) = \{\{3,4\}\}.$$

The faulty server 3 communicates the slices $\mathcal{S}_1(3) = \mathcal{S}_4(3) = \{\{1,3\}\}$ to servers 1 and 4, and the slices $\mathcal{S}_2(3) = \{\{2,3\}\}$ to server 2. The above FBQS induces a subjective quorum system with

$$\mathcal{Q}_1 = \mathcal{Q}_4 = \{\{1,2\},\{1,2,3\},\{1,3,4\},\{1,2,3,4\}\}; \quad \mathcal{Q}_2 = \{\{1,2\},\{1,2,3\},\{1,2,3,4\}\}.$$

Thus, due to server 3 equivocating, servers 1 and 4 consider $\{1,3,4\}$ a quorum, but server 2 does not.

We next adapt the notion of intact servers to subjective FBQSes by requiring the conditions satisfied by intact servers to hold in any subjective view. Given a subjective FBQS $\{\mathcal{S}_v\}_{v\in\mathbf{V}_{\text{ok}}}$, the set of *intact servers* $\mathbf{V}_{\text{int}}$ is the biggest set such that

- all servers in $\mathbf{V}_{\text{int}}$ are correct: $\mathbf{V}_{\text{int}} \cap \mathbf{V}_{\text{bad}} = \emptyset$;
- if $\mathbf{V}_{\text{int}}$ is non-empty, then it is a quorum in $\mathcal{S}_v$ for each $v \in \mathbf{V}_{\text{ok}}$;
- for each $v \in \mathbf{V}_{\text{ok}}$, $\mathcal{S}_v|_{\mathbf{V}_{\text{int}}}$ enjoys quorum intersection.

Note that the definition of intact servers depends only on the slices of correct servers in $\{\mathcal{S}_v\}_{v\in\mathbf{V}_{\text{ok}}}$, which agree across the views $\mathcal{S}_v$ of different servers v. Hence, the notions of intact computed from the subjective views of every server coincide with the above notion.

▶ **Proposition 20.** *The set of intact servers in a subjective FBQS* $\{\mathcal{S}_v\}_{v\in\mathbf{V}_{\rm ok}}$ *with quorum intersection is well-defined and is equal to the set of intact servers in any of* $\mathcal{S}_v$*.*

For instance, the set of intact servers in any of the subjective views in Example 19 is $\{1, 2\}$. As before, servers that are not intact are called befouled.

Using the fact that the views of different servers agree on the quorum slices of correct servers, we can prove the following proposition, which ensures that the intersection of any two quorums, even when coming from the views of different servers, has to contain an intact node. This proposition generalises Proposition 9 and is key for proving the correctness of the Stellar broadcast in the subjective setting.

▶ **Proposition 21.** *Let* $\{\mathcal{S}_v\}_{v\in\mathbf{V}_{\rm ok}}$ *be a subjective FBQS with quorum intersection and at least one intact server, and let* $\{\mathcal{Q}_v\}_{v\in\mathbf{V}_{\rm ok}}$ *be its induced subjective quorum system. Then the intersection of every two quorums in* $\bigcup_{v\in\mathbf{V}_{\rm ok}} \mathcal{Q}_v$ *contains some intact server.*

For example, any two quorums in the quorum system from Example 19 intersect at 1, which is intact.

We can run the Stellar broadcast (with the changes to lines 9, 12 and 15; §4) over a subjective FBQS $\{\mathcal{S}_v\}_{v\in\mathbf{V}_{\rm ok}}$ by letting each correct server v act according to its FBQS $\mathcal{S}_v$, which it constructs based on the information provided by its peers. In particular, a set $B \subseteq \mathbf{V}$ is now considered v-blocking when it overlaps every slice in $\mathcal{S}_v(v)$: $\forall q \in \mathcal{S}_v(v).\, q \cap B \neq \emptyset$. In this setting, the Stellar broadcast still implements weakly reliable Byzantine broadcast, defined as before using the above notion of intact.

▶ **Theorem 22.** *Given sets of correct and faulty servers,* $\mathbf{V}_{\rm ok}$ *and* $\mathbf{V}_{\rm bad}$*, let* $\{\mathcal{S}_v\}_{v\in\mathbf{V}_{\rm ok}}$ *be a subjective FBQS with quorum intersection and at least one intact server. Then the Stellar broadcast over this system implements weakly reliable Byzantine broadcast.*

We can also establish an analogue of Theorem 15, showing that, when servers accept quorums they do not belong to, the broadcast guarantees are strengthened.

The above theorem is our key result: it shows that the Stellar broadcast is correct in the setting of its intended deployment, where servers may have inconsistent information about the trust choices of their peers. This setting has not been considered by the existing correctness theorems for Stellar [11]. Note that, like in the idealised setting we considered earlier, a server in the Stellar broadcast is able to operate when knowing only the quorum slices of the servers it directly interacts with (even though those reported by faulty servers may be bogus). Theorem 22 holds because the key lemmas used to prove the correctness of the Stellar broadcast in the idealised setting can be adjusted to the more general setting we consider here. The following lemma generalises Lemmas 16 and is used to prove Consistency.

▶ **Lemma 23.** *Assume a subjective FBQS* $\{\mathcal{S}_v\}_{v\in\mathbf{V}_{\rm ok}}$ *with quorum intersection and consider an execution of Stellar broadcast over* $\{\mathcal{S}_v\}_{v\in\mathbf{V}_{\rm ok}}$*. If intact servers send messages* $\texttt{READY}(a)$ *and* $\texttt{READY}(a')$*, then* $a = a'$*.*

We prove the above lemma similarly to Lemma 16, but using Proposition 21 instead of Proposition 9. We prove Totality similarly to the idealised case using the following lemma, generalising Lemma 17.

▶ **Lemma 24.** *Let* $\{\mathcal{S}_v\}_{v\in\mathbf{V}_{\rm ok}}$ *be a subjective FBQS with quorum intersection. Consider an execution of the Stellar broadcast with the set of intact servers in* $\{\mathcal{S}_v\}_{v\in\mathbf{V}_{\rm ok}}$ *being* $\mathbf{V}_{\rm int} \neq \emptyset$*. Assume that* $\mathbf{V}_{\rm int} = V^+ \uplus V^-$ *and for some quorum* U *known by a server* v *we have* $U \cap \mathbf{V}_{\rm int} \subseteq V^+$*. Then either* $V^- = \emptyset$ *or there exists some server* $v' \in V^-$ *such that* V^+ *is* v'*-blocking.*

6 Subjective Dissemination Quorum Systems

We have shown that a federated system where servers may lie about their trust choices generates a system in which every server has its own subjective notion of a quorum. We now investigate such subjective quorum structures independently from the way they are generated, which may be useful in the future for dealing with approaches to decentralising trust different from Stellar. To this end, we introduce the notion of a *subjective Byzantine quorum system*, where different servers may have different Byzantine quorum systems.

Let us fix the sets of correct and faulty servers, $\mathbf{V}_{\text{ok}}$ and $\mathbf{V}_{\text{bad}}$. A *subjective Byzantine quorum system* is a pair of two indexed families – a subjective quorum system $\{\mathcal{Q}_v\}_{v\in\mathbf{V}_{\text{ok}}}$ and a *subjective fail-prone system* $\{\mathcal{B}_v\}_{v\in\mathbf{V}_{\text{ok}}}$, where each $\mathcal{B}_v$ is a fail-prone system. A subjective BQS $(\{\mathcal{Q}_v\}_{v\in\mathbf{V}_{\text{ok}}}, \{\mathcal{B}_v\}_{v\in\mathbf{V}_{\text{ok}}})$ is a *subjective DQS* when the following properties hold:

- *SD-safety:* $\forall v \in \mathbf{V}_{\text{ok}}.\, \exists B \in \mathcal{B}_v.\, \mathbf{V}_{\text{bad}} \subseteq B$;
- *SD-consistency:*
  $\forall v \in \mathbf{V}_{\text{ok}}.\, \forall U_1 \in \mathcal{Q}_v.\, \forall U_2 \in \bigcup_{v'\in\mathbf{V}_{\text{ok}}} \mathcal{Q}_{v'}.\, \forall B \in \mathcal{B}_v.\, \mathbf{V}_{\text{bad}} \subseteq B \implies U_1 \cap U_2 \not\subseteq B$; and
- *SD-availability:* each pair $(\mathcal{Q}_v, \mathcal{B}_v)$ with $v \in \mathbf{V}_{\text{ok}}$ satisfies D-availability.

SD-safety requires that the fail-prone system of each server contains some element that includes all faulty servers. Such a condition was not part of the previous DQS definition, but we required it when stating protocol correctness in a particular execution where the set of faulty servers was fixed (Theorem 3). Since a subjective DQS is already defined in the context of a fixed set of faulty servers, we add this condition to its definition. SD-consistency requires that the intersection of a quorum known by a server v with a quorum known by any other server cannot be completely faulty according to v's view. To formulate this we consider only sets $B \in \mathcal{B}_v$ that contain all faulty servers; at least one such set is guaranteed to exist by SD-safety. This requirement ensures that we can generate a subjective DQS from a subjective FBQS from a view of each correct server as per the mapping in Theorem 10.

▶ **Theorem 25.** *Given sets of correct and faulty servers, $\mathbf{V}_{\text{ok}}$ and $\mathbf{V}_{\text{bad}}$, let $\{\mathcal{S}_v\}_{v\in\mathbf{V}_{\text{ok}}}$ be a subjective FBQS with quorum intersection and with at least one intact server, and let $\{\mathcal{Q}_v\}_{v\in\mathbf{V}_{\text{ok}}}$ be the subjective quorum system it induces. For each $v \in \mathbf{V}_{\text{ok}}$ let $\mathcal{B}_v$ consist of the maximal sets B such that the failure of B leaves the set of intact servers in $\mathcal{Q}_v$ non-empty. Then $(\{\mathcal{Q}_v\}_{v\in\mathbf{V}_{\text{ok}}}, \{\mathcal{B}_v\}_{v\in\mathbf{V}_{\text{ok}}})$ is a subjective DQS.*

The properties of subjective DQSes are strong enough to execute Bracha broadcast assuming each correct server acts according to its view in an arbitrary subjective DQS.

▶ **Theorem 26.** *Given sets of correct and faulty servers, $\mathbf{V}_{\text{ok}}$ and $\mathbf{V}_{\text{bad}}$, let $(\{\mathcal{Q}_v\}_{v\in\mathbf{V}_{\text{ok}}}, \{\mathcal{B}_v\}_{v\in\mathbf{V}_{\text{ok}}})$ be a subjective DQS. Then Bracha broadcast over this system implements reliable Byzantine broadcast.*

In the special case when all views of correct servers are the same, a subjective DQS degenerates into a variant of a usual DQS $(\mathcal{Q}, \mathcal{B})$ that satisfies a weaker version of D-consistency that considers only sets $B \in \mathcal{B}$ that include all faulty servers:

$$\forall U_1, U_2 \in \mathcal{Q}.\, \forall B \in \mathcal{B}.\, \mathbf{V}_{\text{bad}} \subseteq B \implies U_1 \cap U_2 \not\subseteq B.$$

Theorem 26 implies that Bracha broadcast is also correct when executed over this DQS variant, which weakens the classical conditions required for correctness. The following stronger version of SD-consistency does specialise to D-consistency of DQSes:

$$\forall v \in \mathbf{V}_{\text{ok}}.\, \forall U_1 \in \mathcal{Q}_v.\, \forall U_2 \in \bigcup_{v'\in\mathbf{V}_{\text{ok}}} \mathcal{Q}_{v'}.\, \forall B \in \mathcal{B}_v.\, U_1 \cap U_2 \not\subseteq B.$$

Obviously, Bracha broadcast can also be correctly executed over a subjective DQS satisfying this property. Finally, Theorems 25 and 26 imply that Bracha broadcast can be executed over a subjective DQS induced by a subjective FBQS.

▶ **Corollary 27.** *Given sets of correct and faulty servers,* $\mathbf{V}_{\text{ok}}$ *and* $\mathbf{V}_{\text{bad}}$*, let* $\{\mathcal{S}_v\}_{v\in\mathbf{V}_{\text{ok}}}$ *be a subjective FBQS with quorum intersection where at least one server is intact. Let* $(\{\mathcal{Q}_v\}_{v\in\mathbf{V}_{\text{ok}}}, \{\mathcal{B}_v\}_{v\in\mathbf{V}_{\text{ok}}})$ *be the subjective DQS that this FBQS induces. Then Bracha broadcast over this subjective DQS implements reliable Byzantine broadcast.*

Note that constructing the fail-prone system $\mathcal{B}_v$ required to run Bracha broadcast at server v requires the server to have a complete (but possibly imperfect) information about the quorum slices of all servers in the system. Hence, the advantage of the Stellar broadcast over Bracha broadcast is preserved in the subjective setting.

7 Related Work

Byzantine quorum systems were proposed by Malkhi and Reiter [10], who also demonstrated how they can be used to implement read-write registers. The generalisation of Bracha broadcast to dissemination quorum systems in Algorithm 1 follows the techniques used in [10]. Researchers have also considered quorum systems that provide stronger guarantees than dissemination ones and studied which quorum systems yield the best performance characteristics [14]. An interesting avenue for future work is to investigate whether other types of quorum systems can be useful in the federated setting.

The Stellar broadcast as presented in this paper is based on the federated voting protocol, which is crucial to the Stellar consensus. In the Stellar whitepaper [11] this protocol is formulated as a form of binary consensus without termination guarantees. The whitepaper does not establish its relationship to reliable Byzantine broadcast or existing protocols. It does contain some correctness statements for the federated voting protocol; in particular, our proof of Totality (Lemma 17) follows the ideas in the whitepaper. However, the existing proofs only consider the idealised setting where Byzantine servers do not equivocate about their trust choices, which does not reflect the intended deployment of Stellar. They furthermore establish weaker safety and liveness properties than us.

8 Conclusions and Future Work

In this paper we have rigorously studied the basic concepts underlying the Stellar blockchain. In particular, we have established a formal correspondence between federated and classical Byzantine quorum systems, and between the federated voting protocol of Stellar and Bracha protocol for reliable Byzantine broadcast. Our results explicate the main benefit of designing a broadcast protocol tailored for the federated setting: allowing servers to operate based on incomplete information. By formalising the relationship between Stellar and Bracha broadcasts, we were additionally able to show that a variant of the Stellar broadcast closer to the original Bracha broadcast still allows servers to operate based on incomplete information, yet provides stronger guarantees.

We have also faithfully captured a realistic setting for deploying Stellar where Byzantine nodes may equivocate about their trust choices. We have shown that this setting motivates a generalisation of the classical Byzantine quorum systems that allows nodes to have inconsistent information about quorums. This may be useful in the future for dealing with alternative approaches to decentralising trust.

Our results demonstrate that the purpose of FBQSes is not limited to solving consensus, for which they were originally proposed. In particular, we believe that apart from broadcast abstractions, FBQSes can also be used to implement read-write registers on the lines of existing constructions [10, 1]. Even though in this paper we did not handle the whole of Stellar consensus protocol, in the future our results should also enable a rigorous analysis of this protocol and similar ones, such as Ripple [13]. Finally, we hope that the connections we have established between classical broadcast abstractions and the novel Stellar protocols will enable transferring ideas from the existing optimised protocols for BFT consensus [9, 6] into the federated setting.

References

1. Ittai Abraham, Gregory V. Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine disk Paxos: optimal resilience with Byzantine shared memory. *Distributed Computing*, 18(5):387–408, 2006.
2. Gabriel Bracha. Asynchronous Byzantine Agreement Protocols. *Information and Computation*, 75(2):130–143, 1987.
3. Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2nd ed.)*. Springer, 2011.
4. Christian Cachin and Marko Vukolic. Blockchain Consensus Protocols in the Wild (Keynote Talk). In *International Symposium on Distributed Computing (DISC)*, pages 1:1–1:16, 2017.
5. Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, 2002.
6. Allen Clement, Edmund L. Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In *Symposium on Networked Systems Design and Implementation (NSDI)*, pages 153–168, 2009.
7. Cynthia Dwork and Moni Naor. Pricing via Processing or Combatting Junk Mail. In *International Cryptology Conference on Advances in Cryptology (CRYPTO)*, pages 139–147, 1993.
8. Álvaro García-Pérez and Alexey Gotsman. Federated Byzantine Quorum Systems (Extended Version). *CoRR*, 2018. `arXiv:1811.03642`.
9. Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. In *Symposium on Operating Systems Principles (SOSP)*, pages 45–58, 2007.
10. Dahlia Malkhi and Michael K. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.
11. David Mazières. The Stellar Consensus Protocol: A Federated Model for Internet-level Consensus, 2016. URL: `https://www.stellar.org/papers/stellar-consensus-protocol.pdf`.
12. Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2009.
13. David Schwartz, Noah Youngs, and Arthur Britto. The Ripple Protocol Consensus Algorithm, 2014. URL: `https://ripple.com/files/ripple_consensus_whitepaper.pdf`.
14. Marko Vukolic. *Quorum Systems: With Applications to Storage and Consensus*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2012.

Characterizing Asynchronous Message-Passing Models Through Rounds

Adam Shimi

IRIT – Université de Toulouse, 2 rue Camichel, F-31000 Toulouse, France
http://www.irit.fr
adam.shimi@irit.fr

Aurélie Hurault

IRIT – Université de Toulouse, 2 rue Camichel, F-31000 Toulouse, France
http://www.irit.fr
aurelie.hurault@irit.fr

Philippe Quéinnec

IRIT – Université de Toulouse, 2 rue Camichel, F-31000 Toulouse, France
http://www.irit.fr
philippe.queinnec@irit.fr

Abstract

Message-passing models of distributed computing vary along numerous dimensions: degree of synchrony, kind of faults, number of faults... Unfortunately, the sheer number of models and their subtle distinctions hinder our ability to design a general theory of message-passing models. One way out of this conundrum restricts communication to proceed by round. A great variety of message-passing models can then be captured in the Heard-Of model, through predicates on the messages sent in a round and received during or before this round. Then, the issue is to find the most accurate Heard-Of predicate to capture a given model. This is straightforward in synchronous models, because waiting for the upper bound on communication delay ensures that all available messages are received, while not waiting forever. On the other hand, asynchrony allows unbounded message delays. Is there nonetheless a meaningful characterization of asynchronous models by a Heard-Of predicate?

We formalize this characterization by introducing Delivered collections: the collections of all messages delivered at each round, whether late or not. Predicates on Delivered collections capture message-passing models. The question is to determine which Heard-Of predicates can be generated by a given Delivered predicate. We answer this by formalizing strategies for when to change round. Thanks to a partial order on these strategies, we also find the "best" strategy for multiple models, where "best" intuitively means it waits for as many messages as possible while not waiting forever. Finally, a strategy for changing round that never blocks a process forever implements a Heard-Of predicate. This allows us to translate the order on strategies into an order on Heard-Of predicates. The characterizing predicate for a model is then the greatest element for that order, if it exists.

2012 ACM Subject Classification Theory of computation → Distributed computing models

Keywords and phrases Message-passing, Asynchronous Rounds, Dominant Strategies, Failures

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.18

Related Version A full version of the paper is available at [14], https://arxiv.org/abs/1805.01657.

Funding This work was supported by project PARDI ANR-16-CE25-0006.

© Adam Shimi, Aurélie Hurault, and Philippe Quéinnec;
licensed under Creative Commons License CC-BY

22nd International Conference on Principles of Distributed Systems (OPODIS 2018).
Editors: Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira; Article No. 18; pp. 18:1–18:17

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

1 Introduction

1.1 Motivation

Even when restricted to message-passing, distributed computing spawns a plethora of models: with various degrees of synchrony, with different kinds of faults, with different failure detectors... Although some parameters are quantitative, such as the number of faults, the majority are qualitative instead, for example the kinds of faults. Moreover, message-passing models are usually defined by a mix of mathematical formalism and textual description, with crucial details nested deep inside the latter. This is why these models resist unification into a theory of distributed computing, and why results in the field are notoriously hard to organize, use and extend.

One solution requires constraining communication to proceed by round: each process repeatedly broadcasts a message with its current round number, waits for as many messages as possible bearing this round number, and changes round by computing both its next state and next message. The variations between models are then captured by the dynamic graph specifying, for each round, from which processes each process received a message with this round number before the end of its round; this fits the concept of dynamic network from Kuhn and Oshman [11]. Nonetheless, we will privilege the perspective of Charron-Bost and Schiper Heard-Of model [4], which places itself more at the level of processes. Here, the Heard-Of collection of an execution contains, for each round r and each process j, the set of processes from which j received a message sent in round r before going to round $r + 1$. Then, a predicate on Heard-Of collections characterizes a message-passing model.

Yet rounds don't remove the complexities and subtleties of message-passing models − they just shift them to the characterization of a given model by a Heard-Of predicate. This characterization depends on how rounds can be implemented in the underlying model. In the synchronous case, processes progress in lock-step, and every message that will ever be received is received during its corresponding round. Hence, the Heard-Of predicate characterizing a synchronous model simply specifies which messages can be lost. In asynchronous models on the other hand, messages can be late, and thus the distance between the round numbers of processes is unbounded. The combination of these uncertainties implies that processes do not know which messages will be delivered and when. Thus, there is a risk of not waiting for useful messages that will eventually arrive, and to wait forever for messages that never will.

To the best of our knowledge, there is no systematic study of the Heard-Of predicates generated by various asynchronous message-passing models. Because it is a crucial step in unifying distributed computing's menagerie of models through rounds, we also believe this topic to be of importance.

1.2 Approach and Overview

As hinted above, the difficulty lies in the potential discrepancy between messages delivered on time − captured by Heard-Of collections − and messages delivered at all. We want to determine the former, but it is considerably easier to specify the latter for an operational model: list the messages that will eventually arrive. We therefore center our formalization around Delivered collections, infinite sequences of communication graphs capturing, for each round, all messages sent at this round and eventually delivered for a given operational model. The question is thus to characterize by a Heard-Of predicate which messages can be waited for when the deliveries are those from the Delivered predicate.

From these Delivered collections, we build runs representing the different scheduling of deliveries and changes of round. Some of these runs, called valid, define a Heard-Of collection; invalid runs have processes blocked forever at some round. We filter the latter thanks to strategies: sets of local states for which a process is allowed to change round. Runs for a strategy must also satisfy a fairness condition ensuring that if a process can change round continuously, it does. Strategies with only valid runs for a Delivered predicate, that is strategy implementing a Heard-Of predicate, are called valid.

The next question is how to choose a valid strategy and the corresponding predicate, as characterizing a Delivered predicate and its underlying model? We answer by taking the strategy generating the Heard-Of predicate that is the smallest overapproximation of the Delivered predicate. From this intuition, we define a partial order on valid strategies called domination; a characterizing strategy is a greatest element for this order, and the characterizing predicate is the one it generates.

The results obtained with this approach are threefold:

- The formalization itself, with a complete example: the asynchronous message-passing model with reliable communication and at most F permanent crashes.
- The study of carefree strategies, the ones depending only on messages from the current round. This restricted class is both well-behaved enough to always have a unique dominating strategy, and expressive enough to capture interesting Delivered predicates.
- The study of reactionary strategies, the ones depending only on messages from past and current rounds. Here too we show well-behavior of this class as well as an example where reactionary is needed for domination, and another one where it is insufficient.

Along these results, we also formally prove the characterization of asynchronous models by Heard-Of predicates given by Charron-Bost and Schiper [4].

We begin by the formalization in **Section 2**, while **Section 3** introduces a fully developed example: the asynchronous message-passing model with reliable communication and at most F permanent crashes. **Section 4** explores carefree strategies in terms of well-behavior and expressivity, closing with the example with at most B failed broadcasts per round, a Delivered predicate dominated by a carefree strategy. We follow by studying reactionary strategies in **Section 5**. Here again, well-behavior and expressivity are examined, followed by the example of at most F permanent initial crashes. **Section 6** and **Section 7** then conclude the paper with a discussion of related works, the value of our results and some perspectives.

Due to space constraints, we only provide sketches for some proofs; the complete versions can be found in the long paper [14].

2 Formalization

All our abstractions revolve around infinite sequences of graphs, called collections. A Delivered collection maps each round r and process j to the set of processes from which j receive a message sent at r. A Heard-Of collection maps each round r and process j to the set of processes from which j received, before going to round $r+1$, the message sent at r. The difference lies in considering all deliveries for Delivered collections, but only the deliveries before the end of the round of the receiver for Heard-Of collections.

▶ **Definition 1** (Collections and Predicates)**.** Let Π a set of processes. $Col : (\mathbb{N}^* \times \Pi) \mapsto \mathcal{P}(\Pi)$ is either a **Delivered collection** or a **Heard-Of collection** for Π, depending on the context.

In the same way, $Pred : \mathcal{P}((\mathbb{N}^* \times \Pi) \mapsto \mathcal{P}(\Pi))$ is either a **Delivered predicate** or a **Heard-Of predicate** for Π.

For a given *Col*, the kernel of round r are the processes from which everyone receives a message for this round $K_{Col}(r) \triangleq \bigcap_{j \in \Pi} Col(r, j)$.

2.1 Runs and Strategies

The behavior of processes is classically specified by runs, sequences of both states and transitions satisfying some restricting conditions: messages cannot be delivered before the round they are sent and are delivered only once. Given a Delivered collection, we additionally require that the delivered messages are exactly the ones in the Delivered collection.

As we only care about which messages can be waited for, ignoring the content of messages or the underlying computation, we limit the state to the received messages and the round.

▶ **Definition 2** (Run). Let Π be a set of n processes. Let $Q = (\mathbb{N} \times \mathcal{P}(\mathbb{N}^* \times \Pi))$ the set of process states. The first element is the round of a process (written $q.round$ for $q \in Q$) and the second is the set of pairs $\langle$round it was sent, sender$\rangle$ for each delivered message (written $q.received$ for $q \in Q$). Let the set of transitions $T = \{next_j \mid j \in \Pi\} \cup \{deliver(r, k, j) \mid r \in \mathbb{N}^* \wedge k, j \in \Pi\} \cup \{end\}$. $next_j$ is the transition for j changing round, $deliver(r, k, j)$ is the transition for the delivery to j of the message sent by k in round r, and end is the transition to end a finite run. For $qt \in Q^n \times T$ and $j \in \Pi$, we write $qt.state$ for the state, $qt.state.j$ for the local state of j and $qt.transition$ for the transition. Finally, let $(Q^n \times T)^\infty$ be the set of finite and infinite words on the set $(Q^n \times T)$. Then, $t \in (Q^n \times T)^\infty$ is a **run** $\triangleq$

- **(Initial state)** $t[0].state = \langle 1, \emptyset \rangle^n$
- **(Transitions)** $\forall i \in [0, size(t))$:

$$\begin{pmatrix} \exists r \in \mathbb{N}^*, \exists k, j \in \Pi : t[i].transition = deliver(r, k, j) \implies \\ \quad t[i+1].state = t[i].state \\ \qquad\qquad Except\ t[i+1].state.j.received = t[i].state.j.received \cup \{(r, k)\} \\ \wedge\ \exists j \in \Pi : t[i].transition = next_j \implies \\ \quad t[i+1].state = t[i].state\ Except\ t[i+1].state.j.round = t[i].state.j.round + 1 \\ \wedge\ (t[i].transition = end) \implies (i = size(t) - 1) \end{pmatrix}$$

- **(Delivery after sending)** $\forall i \in [0, size(t))$: $t[i].transition = deliver(r, k, j) \implies t[i].state.k.round \geq r$
- **(Unique delivery)** $\forall \langle r, k, j \rangle \in (\mathbb{N}^* \times \Pi \times \Pi) : (\exists i \in [0, size(t)) : t[i].transition = deliver(r, k, j)) \implies (\forall i' \in [0, size(t)) \setminus \{i\} : t[i'].transition \neq deliver(r, k, j))$

Let *CDel* be a Delivered collection. Then, $runs(CDel)$, the **runs of** *CDel* $\triangleq$

$$\left\{ t\ a\ run \;\middle|\; \begin{array}{l} \forall \langle r, k, j \rangle \in \mathbb{N}^* \times \Pi \times \Pi : \\ \quad (k \in CDel(r, j) \wedge \exists i \in [0, size(t)) : t[i].state.j.round \geq r) \\ \quad \iff \\ \quad (\exists i \in [0, size(t)) : t[i].transition = deliver(r, k, j)) \end{array} \right\}$$

For *PDel* a Delivered predicate, we write $runs(PDel) = \{runs(CDel) \mid CDel \in PDel\}$.

Our definition of runs does not force processes to change rounds. This contradicts our intuition about a system using rounds: processes should keep on "forever", or at least as long as necessary. In a valid run, processes change round an infinite number of times.

▶ **Definition 3** (Validity). A run t is **valid** $\triangleq \forall j \in \Pi : |\{i \in \mathbb{N} \mid t[i].transition = next_j\}| = \aleph_0$.

Valid run are necessarily infinite. Yet the definition above allows finite runs thanks to the *end* transition. This is used in proofs by contradiction which imply the manipulation of invalid runs and thus potentially finite ones.

Next, we define the other building block of our approach: strategies. They are simply sets of local states, representing the states where processes can change round.

▶ **Definition 4** (Strategy). $f : \mathcal{P}(Q)$ is a **strategy**.

Combining a Delivered predicate and a strategy results in runs capturing the behavior of processes for the corresponding Delivered collections when following the strategy. In these runs, processes can change round only when allowed by the strategy, and must also do so if the strategy allows it continuously.

▶ **Definition 5** (Runs Generated by a Strategy). Let f be a strategy and t a run. t is a **run generated by** $f \triangleq t$ satisfies the following:

- **(Next only if allowed)** $\forall i \in [0, size(t)), \forall j \in \Pi : (t[i].transition = next_j \implies t[i].state.j \in f)$
- **(Infinite fairness of next)** If t is infinite, then $\forall j \in \Pi : |\{i \in \mathbb{N} \mid t[i].transition = next_j\}| < \aleph_0 \implies |\{i \in \mathbb{N} \mid t[i].state.j \notin f\}| = \aleph_0$
- **(Finite fairness of next)** If t is finite, then $\forall j \in \Pi : t[size(t) - 1].state.j \notin f$.

For a Delivered predicate $PDel$, we note $runs_f(PDel) = \{t \text{ a run} \mid t \text{ generated by } f \wedge t \in runs(PDel)\}$.

From that point, it is clear that a well-behaved strategy is such that all its runs are valid.

▶ **Definition 6** (Valid Strategy). Let $PDel$ a Delivered predicate and f a strategy. f is a **valid strategy** for $PDel \triangleq \forall t \in runs_f(PDel) : t$ is a valid run.

Validity guarantees an infinite number of complete rounds for every run of the strategy. This ensures that a run defines a Heard-Of collection, as we see next.

2.2 From Delivered Collections to Heard-Of Collections

Recall that the difference between a Heard-Of and a Delivered collection is that the latter takes into account all delivered messages, while the former only considers messages from a round if they were received before or during the corresponding round of the receiver.

If a run is valid, then all processes have infinitely many rounds, and thus it defines a Heard-Of collection through its behavior.

▶ **Definition 7** (Heard-Of Collection of a Valid Run). Let t a valid run. Then, CHO_t is the **Heard-Of collection of** $t \triangleq$

$$\forall r \in \mathbb{N}^*, \forall j \in \Pi : CHO_t(r, j) = \left\{ k \in \Pi \;\middle|\; \exists i \in \mathbb{N} : \begin{pmatrix} & t[i].state.j.round = r \\ \wedge & t[i+1].j.state.round = r + 1 \\ \wedge & \langle r, k \rangle \in t[i].state.j.received \end{pmatrix} \right\}$$

It is useful to go the other way, and extract from a Heard-Of collection some canonical valid run generating it. Our choice is a run where processes change round in lockstep, every message from a Heard-Of set is delivered in the round where it was sent, and every late message is delivered in the round following the one where it was sent.

▶ **Definition 8** (Standard Run of a Heard-Of collection). Let cho be a Heard-Of collection. For $r > 0$, let $onTimeMes_r$ be a permutation of $\{deliver(r, k, j) \mid k, j \in \Pi \wedge k \in cho(r, j)\}$, $lateMes_r$ a permutation of $\{deliver(r - 1, k, j) \mid k, j \in \Pi \wedge k \notin cho(r - 1, j)\}$, and $nexts$ be a permutation of $\{next_j \mid j \in \Pi\}$. Then, the run starting at the initial state and with transitions defined by the word $\prod_{r>0} (lateMes_r.onTimeMes_r.nexts)$ is a **standard run** of cho.

This canonical run is a run of any Delivered predicate containing the collection where every message is delivered. This collection captures the case where no failure occurs: every process always broadcasts and no message is lost. Having this collection in a Delivered predicate ensures that although faults might happen, they are not forced to do so.

▶ **Lemma 9** (Standard Run is a Run for Total Collection). *Let cho be a Heard-Of collection and let PDel be a Delivered predicate containing the total collection $CDel_{total}$ defined by $\forall r > 0, \forall j \in \Pi : CDel_{total}(r, j) = \Pi$. Then, a standard run t of cho is a run of PDel.*

Proof. t is a run since it satisfies the four constraints defining a run. Furthermore, one message from each process is eventually delivered to everyone for each round in t, which means that t is a run of the total Delivered collection, and thus a run of *PDel*. ◀

▶ **Definition 10** (Heard-Of Predicate Generated by Strategy). If f is a valid strategy for *PDel*, we write $PHO_f(PDel)$ for the Heard-Of collections of the runs generated by f for *PDel*: $PHO_f(PDel) \triangleq \{CHO_t \mid t \in runs_f(PDel)\}$.

Every valid strategy generates a Heard-Of predicate from the Delivered predicate. We now have a way to go from a Delivered predicate to a Heard-Of one: design a valid strategy for the former that generates the latter. But we still have not answered the original question: among all the Heard-Of predicates one can generate from a given Delivered predicate, which one should we consider as the characterization of the Delivered predicate?

First, remark that all Delivered collections can be generated as Heard-Of collections by a valid strategy: simply deliver all messages from a round before changing the round of a process – the change of round must eventually happen by validity of the strategy. Thus, every Heard-Of predicate generated from a Delivered one is an overapproximation of the latter: for any strategy f, $PDel \subseteq PHO_f(PDel)$. But we want to receive as many messages as possible on time, that is to be as close as possible to the original Delivered predicate. The characterizing Heard-Of predicate is thus the smallest such overapproximation, if it exists.

We formalize this intuition by defining a partial order on valid strategies for a Delivered predicate capturing the implication of the generated Heard-Of predicates. One strategy dominates another if the Heard-Of set it generates is included in the one generated by the other. Dominating strategies are then the greatest elements for this order. By definition of domination, all dominating strategies generate the same dominating Heard-Of predicate, which characterizes the Delivered predicate.

▶ **Definition 11** (Domination Order, Dominating Strategy and Dominating Predicate). Let *PDel* be a Delivered predicate and let f and f' be two valid strategies for *PDel*. Then, f **dominates** f' for *PDel*, written $f' \prec_{PDel} f \triangleq PHO_{f'}(PDel) \supseteq PHO_f(PDel)$.

A greatest element for $\prec_{PDel}$ is called a **dominating strategy** for *PDel*. Given such a strategy f, the **dominating predicate** for *PDel* is then $PHO_f(PDel)$.

3 A Complete Example: At Most F Crashes

To provide a more concrete intuition, we turn to an example: the message-passing model with asynchronous and reliable communication, and at most F permanent crashes. To find the corresponding Delivered predicate, we characterize which messages are delivered in a round execution of this model: all the messages sent by a process before it crashes are delivered; it sends no message after; at the round where it crashes, there may be an incomplete broadcast if the process crashes in the middle of it. Additionally, at most F processes can crash.

▶ **Definition 12** ($PDel^F$). The Delivered predicate $PDel^F$ for the asynchronous model with reliable communication and at most F permanent crashes $\triangleq$

$$\left\{ CDel \in (\mathbb{N}^* \times \Pi) \mapsto \mathcal{P}(\Pi) \;\middle|\; \forall r > 0, \forall j \in \Pi : \begin{array}{ll} & |CDel(r,j)| \geq n - F \\ \wedge & CDel(r+1,j) \subseteq K_{CDel}(r) \end{array} \right\}.$$

The folklore strategy for this model is to wait for at least $n - F$ messages before allowing the change of round.

▶ **Definition 13** (waiting for $n - F$ messages)**.** The strategy to wait for $n - F$ messages is: $f_{n-F} \triangleq \{q \in Q \mid |\{k \in \Pi \mid \langle q.round, k\rangle \in q.received\}| \geq n - F\}$

To see why this strategy is used in the literature, simply remark that at least $n - F$ messages must be delivered to each process at each round. Thus, waiting for that many messages ensures that no process is ever blocked. Rephrased with the concepts introduced above, f_{n-F} is a valid strategy for $PDel^F$.

▶ **Lemma 14** (Validity of f_{n-F})**.** *f_{n-F} is valid for $PDel^F$.*

Proof. We proceed by contradiction: **Assume** f_{n-F} is invalid for $PDel^F$. Thus, there exists $t \in runs_{f_{n-F}}(PDel^F)$ invalid. Because t is infinite, the problem is either the infinite fairness of *next* or a *next* done when f_{n-F} does not allow it. Each *next* transition being played an infinite number of times, we conclude that some *next* is played while f_{n-F} does not allow it. Let r be the smallest round where it happens and j be a process blocked at r in t. Let also $CDel_t$ be a Delivered collection of $PDel^F$ such that $t \in runs(CDel_t)$.

We know by definition of $PDel^F$ that $|CDel_t(r, j)| \geq n - F$. The minimality of r and the fact that $t \in runs(CDel)$ then ensure that all messages in this Delivered set are delivered at some point in t. By definition of f_{n-F}, the transition $next_j$ is then available from this point on. This **contradicts** the fact that j cannot change round at this point in t. ◀

The Heard-Of predicate generated by f_{n-F} was first given by Charron-Bost and Schiper [4] as a characterization the asynchronous model with reliable communication and at most F crashes. The intuition behind it is that even in the absence of crashes, we can make all processes change round by delivering any set of at least $n - F$ messages to them.

▶ **Theorem 15** (Heard-Of Characterization of f_{n-F})**.**
$PHO_{f_{n-F}}(PDel^F) = \{cho \in (\mathbb{N}^* \times \Pi) \mapsto \mathcal{P}(\Pi) \mid \forall r \in \mathbb{N}^*, \forall j \in \Pi : |cho(r, j)| \geq n - F\}$.

Proof. First, we show $\subseteq$. Let $cho \in PHO_{f_{n-F}}(PDel^F)$ and $t \in runs_{f_{n-F}}(PDel^F)$ a run of f_{n-F} generating *cho*. By definition of the runs of f_{n-F}, processes change round only when they received at least $n - F$ messages from the current round, which implies that $\forall r \in \mathbb{N}^*, \forall j \in \Pi : |cho(r, j)| \geq n - F$.

Then, we show $\supseteq$. Let *cho* a Heard-Of collection over Π such that $\forall r \in \mathbb{N}, \forall j \in \Pi :$ $|cho(r, j)| \geq n - F$. Let t be a standard run of *cho*; since $PDel^F$ contains the total collection, t is a run of $PDel^F$ by Lemma 9. To prove this is also a run of f_{n-F}, we proceed by contradiction: **Assume** it is not: because t is infinite, the problem is either the infinite fairness of *next* or a *next* done when f_{n-F} does not allow it. Each *next* transition being played an infinite number of times in a standard run, the only possibility left is the second one: some *next* transition in t is done while f_{n-F} does not allow the corresponding process to change round. Let r be the smallest round where this happens, and j one of the concerned processes at round r. By definition of t as a standard run, j received all messages from $cho(r, j)$ before the problematic *next*. And $|cho(r, j)| \geq n - F$ by hypothesis. By definition of f_{n-F}, the transition $next_j$ is then available from this point on. This **contradicts** the fact that j cannot change round at this point. We conclude that $cho \in PHO_{f_{n-F}}(PDel^F)$. ◀

Finally, we want to vindicate the folklore intuition about this strategy: that it is optimal in some sense. Intuitively, waiting for more than $n - F$ messages per round means risking waiting forever, and waiting for less is wasteful. Our domination order captures this concept

of optimality: we show that f_{n-F} is indeed a dominating strategy for $PDel^F$. Therefore, $PHO_{f_{n-F}}(PDel^F)$ is the dominating predicate for $PDel^F$.

To do so, we introduce another canonical run, this time for the combination of a Delivered collection and a strategy. This run consists in delivering every message as early as possible.

▶ **Definition 16** (Earliest Run of Strategy for Delivered Collection)**.** Let *CDel* be a Delivered collection and f be a strategy. An **earliest run** of f for *CDel* is a run of $runs_f(CDel)$ starting in the initial state, and with the following transitions happening in successive iterations:

- At each iteration, all messages that can be delivered and were not already are delivered.
- Then, every process allowed by f to change round does so.

By combining standard and earliest runs, we show that any valid strategy for $PDel^F$ is dominated by f_{n-F} and thus that $PHO_{f_{n-F}}(PDel^F)$ is the dominating predicate for $PDel^F$.

▶ **Theorem 17** (f_{n-F} Dominates $PDel^F$)**.** *f_{n-F} dominates $PDel^F$.*

Proof. Let f be a valid strategy for $PDel^F$; we now prove that $f \prec_{PDel^F} f_{n-F}$, that is $PHO_{f_{n-F}}(PDel^F) \subseteq PHO_f(PDel^F)$. Let $cho \in PHO_{f_{n-F}}(PDel^F)$, and let t be a standard run of *cho*. Since $PDel^F$ contains the total collection, t is a run of $PDel^F$ by Lemma 9. We only need to prove that it is also a run of f to conclude.

We do so by contradiction. **Assume** t is not a run of f: because t is infinite, the problem is either the infinite fairness of *next* or a *next* done by a process j when f does not allow it. Each *next* transition being played an infinite number of times in a standard run, the only possibility left is the second one. At the point of the forbidden *next*, by definition of a standard run, j has received every message from previous rounds, and all messages from $cho(r, j)$. By application of Theorem 15 and *cho* being in $PHO_{f_{n-F}}(PDel^F)$, $cho(r, j)$ contains at least $n - F$ processes.

Let $CDel_{block}$ be the Delivered collection where all processes from which j did not receive a message at the problematic *next* in t stop sending messages from this round on:
$$\forall r' > 0, \forall k \in \Pi : CDel_{block}(r', k) = \begin{cases} \Pi & \textit{if } r' < r \\ cho(r, j) & \textit{otherwise} \end{cases}$$

This is a Delivered collection of $PDel^F$: processes that stop sending messages never do again, and at most F processes do so because $cho(r, j)$ contains at least $n - F$ processes.

Let t_{block} be an earliest run of f for $CDel_{block}$. This is a run of f, by definition. We then have two possibilities.

- During one of the first $r - 1$ iterations of t_{block}, there is some process which cannot change round. Let r' be the smallest iteration where it happens, and k be a process unable to change round at this iteration. By minimality of r', all processes arrive at round r', and by symmetry of $CDel_{block}$ they all receive the same messages as k. Thus, all processes are blocked at round r', there are no more next or deliveries, and t_{block} is therefore invalid.
- For the first $r - 1$ iterations, all processes change round. Thus, every one arrives at round r. By definition of an earliest run, all messages from the round are delivered before any *next*. The symmetry of $CDel_{block}$ also ensures that every process received the same messages, that is all messages from round $< r$ and all messages from $cho(r, j)$. These are exactly the messages received by j in t at round r. But by hypothesis, j is blocked in this state in t. We thus deduce that all processes are blocked at round r in t_{block}, and thus that it is an invalid run.

Either way, we deduce that f is invalid, which is a **contradiction**. ◀

This means that when confronted with a model captured by $PDel^F$, there is no point in remembering messages from past rounds – and messages from future rounds are simply buffered. Intuitively, messages from past rounds are of no use in detecting crashes in the current round. As for messages from future rounds, they could serve to detect that a process has not crashed when sending its messages from the current round. This does not alter the Heard-Of predicate because nothing forces messages from future rounds to be delivered early, and thus there is no way to systematically use the information from future rounds.

4 Carefree Strategies

We now turn to more general results about Delivered predicates and strategies. We focus first on a restricted form of strategies, the carefree ones: they depend only on the received messages from the current round. For example, f_{n-F} is a carefree strategy. These are quite simple strategies, yet they can be dominating, as shown for f_{n-F} and $PDel^F$.

4.1 Definition and Expressiveness Results

▶ **Definition 18** (Carefree Strategy). Let f be a strategy and, $\forall q \in Q$, let $cfree(q) = \{k \in \Pi \mid \langle q.round, k\rangle \in q.received\}$. f is a **carefree strategy** $\triangleq \forall q, q' \in Q : cfree(q) = cfree(q') \implies (q \in f \iff q' \in f)$.

For f a carefree strategy, let $Nexts_f \triangleq \{cfree(q) \mid q \in f\}$. It uniquely defines f.

Thus a carefree strategy can be defined by a set of sets of processes: receiving a message from all processes in any of those set makes the strategy authorize the change of round. This gives us a simple necessary condition on such a strategy to be valid: its *Nexts* set must contains all Delivered set from the corresponding Delivered predicate. If it does not, then an earliest run of any collection containing a Delivered set not in the *Nexts* would be invalid.

This simple necessary condition also proves sufficient.

▶ **Lemma 19** (Validity of Carefree). *Let PDel be a Delivered predicate and f a carefree strategy. Then, f is valid for PDel* $\iff \forall CDel \in PDel, \forall r > 0, \forall j \in \Pi : CDel(r, j) \in Nexts_f$.

Proof sketch. ($\Rightarrow$) Let f be valid for *PDel*. We show by contradiction that it is impossible that $\exists CDel \in PDel$ that falsifies the right-hand side. Let t be an earliest run of f for a *CDel*, which is a run of f by definition. If all processes reach round r, then by definition of t, all messages from round r are delivered before any *next* at round r; if a process cannot change round at this point, it will never be able later as its set of received messages from r will never change, and t would be invalid. If some process is stuck at a round $r' < r$ while the other ones reach rounds $\geq r'$, by the same reasoning, it has received all available messages from round r' and will never be able to change round, which makes t invalid. Both cases contradict the validity of f.

($\Leftarrow$) Let *PDel* and f which satisfies the right-hand side, and assume that f is invalid. For an invalid run $t \in runs_f(PDel)$, there is a delivery collection *CDel* and a smallest round r where a process is blocked. By minimality of r, all processes reach r and the blocked process eventually receives all messages from r, and thus can change round, which contradicts the fact that some process is blocked. ◀

Carefree strategies are elegant, but they also have some drawbacks: mainly that the Heard-Of predicates they can implement are quite basic. Precisely, when the Delivered predicate contains the total collection, they implement predicates where the Heard-Of collections are all possible combinations of Delivered sets from the original Delivered predicate.

▶ **Theorem 20** (Heard-Of Predicates of Carefree Strategy). *Let PDel be a Delivered predicate containing the total collection, and let f be a valid carefree strategy for PDel. Then, $\forall cho$ a Heard-Of collection for Π : $cho \in CHO_f(PDel) \iff \forall r > 0, \forall j \in \Pi : cho(r, j) \in Nexts_f$.*

Proof. ($\Rightarrow$) This direction follows from the definition of a carefree strategy: it allows changing round only when the messages received from the current round form a set in its *Nexts*.

($\Leftarrow$) Let *cho* be a Heard-Of collection for Π such that $\forall r > 0, \forall j \in \Pi : cho(r, j) \in Nexts_f$. Let t be a standard run of *cho*. It is a run by Lemma 9. It is also a run of f because at each round, processes receive messages from a set in $Nexts_f$ and are thus allowed by f to change round. We conclude that $cho \in PHO_f(PDel)$. ◀

Finally, carefree strategies always have a dominating element for a given Delivered predicate.

▶ **Theorem 21** (Always a Dominating Carefree Strategy). *Let PDel be a Delivered predicate and let f_{cfDom} be the carefree strategy with $Nexts_f = \{CDel(r, j) \mid CDel \in PDel \land r > 0 \land j \in \Pi\}$. f_{cfDom} dominates all carefree strategies for PDel.*

Proof. First, f_{cfDom} is valid for *PDel* by application of Lemma 19.

As for domination, we also deduce from Lemma 19 that $\forall f$ a valid carefree strategy for *PDel*, $Nexts_{f_{cfDom}} \subseteq Nexts_f$ and thus $f_{cfDom} \subseteq f$. Therefore, $\forall q \in Q : q \in f_{cfDom} \implies q \in f$. This gives us $runs_{f_{cfDom}}(PDel) \subseteq runs_f(PDel)$, and we conclude $PHO_{f_{cfDom}}(PDel) \subseteq PHO_f(PDel)$. Therefore, f_{cfDom} dominates all valid carefree strategies for *PDel*. ◀

In the case where *PDel* allows the delivery of all messages, that is contains the total collection, there is only one carefree strategy which dominates all carefree strategies.

▶ **Theorem 22** (With Total Collection, Unique Dominating Carefree Strategy). *Let PDel be a Delivered predicate containing the total collection. Let f_{cfDom} be the carefree strategy with $Nexts_f = \{CDel(r, j) \mid CDel \in PDel \land r > 0 \land j \in \Pi\}$. f_{cfDom} is the unique carefree strategy which dominates all carefree strategies for PDel.*

Proof. First, f_{cfDom} dominates all carefree strategies for *PDel* by Theorem 21. To show uniqueness, let f be a valid carefree strategy different from f_{cfDom}; thus $Nexts_f \neq Nexts_{f_{cfDom}}$. By Lemma 19, we also have $Nexts_{f_{cfDom}} \subseteq Nexts_f$. Therefore, $Nexts_{f_{cfDom}} \subsetneq Nexts_f$.

Let $D \in Nexts_f \setminus Nexts_{f_{cfDom}}$ and let $cho \in (\mathbb{N}^* \times \Pi) \mapsto \mathcal{P}(\Pi)$ the Heard-Of collection such that $\forall r > 0, \forall j \in \Pi : cho(r, j) = D$. By application of Theorem 20, this is a Heard-Of collection generated by f but not by f_{cfDom}. Therefore, $PHO_f(PDel) \not\subseteq PHO_{f_{cfDom}}(PDel)$, and f' does not dominate f_{cfDom}. ◀

4.2 When Carefree is Enough

Finally, the value of carefree strategy depends on which Delivered predicates have such a dominating strategy. We already know that $PDel^F$ does; we now extend this result to a class of Delivered predicates called Round-symmetric. This condition captures the fact that given any Delivered set D, one can build, for any $r > 0$, a Delivered collection where processes receive all messages up to round r, and then they share D as their Delivered set in round r. As a limit case, the predicate also contains the total collection.

▶ **Definition 23** (Round-Symmetric Delivered Predicate). Let *PDel* be a Delivered Predicate. *PDel* is **round-symmetric** $\triangleq$

- **(Total collection)** $PDel$ contains the total collection: $CDel_{total} \in PDel$ where $CDel_{total}$ is defined by $\forall r > 0, \forall j \in \Pi : CDel_{total}(r, j) = \Pi$.
- **(Symmetry up to a round)** $\forall D \in \{CDel(r, j) \mid CDel \in PDel \wedge r > 0 \wedge j \in \Pi\}$, $\forall r > 0, \exists CDel \in PDel, \forall j \in \Pi : (\forall r' < r : CDel(r', j) = \Pi \wedge CDel(r, j) = D)$

What round-symmetry captures is what makes $PDel^F$ be dominated by a carefree strategy: the inherent symmetry of these Delivered collections allows us to block processes with exactly the same received messages. This allows us to show that any valid strategy should allow changing round at this point, which is fundamental to any proof of domination.

▶ **Theorem 24** (Sufficient Condition of Carefree Domination)**.** *Let PDel be a Round-symmetric Delivered predicate. Then, there is a carefree strategy which dominates PDel.*

Proof Sketch. Let f be the carefree strategy dominating all carefree strategies for $PDel$. It exists by Theorem 21 and, as $PDel$ contains the total collection, it is unique by Theorem 22. Let f' be a valid (not necessarily carefree) strategy for $PDel$. We prove that $f' \prec_{PDel^F} f$, that is $PHO_f(PDel) \subseteq PHO_{f'}(PDel)$. Let $cho \in PHO_f(PDel)$ and t a standard run of cho. We show t is a run of f' by contradiction on the validity of f'. Assume it is not, then there is some $next_j$ transition at some round r in t that f' does not allow.

The round-symmetry of $PDel$ allows us to build $CDel_{block} \in PDel$ such that $\forall r' < r, \forall k \in \Pi : CDel_{block}(r', k) = \Pi$ and $\forall k \in \Pi : CDel_{block}(r, k) = cho(r, j)$. Consider t_{block}, an earliest run of f' for $CDel_{block}$. For a round $r' < r$, all processes get a message from everyone. By symmetry, either all processes are blocked, which makes f' invalid, or all processes change round. For round r, all processes get the same set $cho(r, j)$, and as a result, receive the same messages that prevent j from changing round in t. No process is allowed to change round by f' at this point, which makes f' invalid. ◀

As another example of a Delivered predicate satisfying this condition, we study the model where at most B broadcasts per round can fail: either all processes receive the message sent by a process at a round or none does; there are at most B processes per round that can be in the latter case, where no one receives their message. The Delivered predicate states that each process receives the kernel of the round, and this kernel contains at least $n - B$ processes.

▶ **Definition 25** ($PDel^B$)**.** The Delivered predicate $PDel^B$ corresponding to at most B full broadcast failures is:
$\{CDel \in (\mathbb{N}^* \times \Pi) \mapsto \mathcal{P}(\Pi) \mid \forall r > 0, \forall j \in \Pi : CDel(r, j) = K_{CDel}(r) \wedge |K_{CDel}(r)| \geq n - B\}$.

The astute reader might have noticed that the carefree strategy dominating all carefree for this Delivered predicate is f_{n-B} (which is f_{n-F} with F instantiated to B).

▶ **Lemma 26** (f_{n-B} Carefree Dominates $PDel^B$)**.** *f_{n-B} dominates carefree strategies for $PDel^B$.*

Proof. The definition of $PDel^B$ gives us $\{CDel(r, j) \mid CDel \in PDel \wedge r > 0 \wedge j \in \Pi\} = \{ S \in \mathcal{P}(\Pi) \mid |S| \geq n - B\} = Nexts_{f_{n-B}}$, because the only constrained on the Delivered sets are that they must be the same for all processes at a round and that they must have at least $n - B$ processes. We therefore conclude from Lemma 19 that f_{n-B} is valid for $PDel^B$ and from Theorem 21 that it dominates carefree strategy for the same predicate. ◀

▶ **Theorem 27** (f_{n-B} Dominates $PDel^B$)**.** *f_{n-B} dominates $PDel^B$.*

Proof. We only need to prove that $PDel^B$ is round-symmetric; Theorem 24 will then yield that $PDel^B$ is dominated by a carefree strategy, thus dominated by the carefree strategy that dominates all carefree strategies. First, $PDel^B$ contains the total collection where no failure occurs. Then, let D be any $CDel(_,_)$ of $PDel^B$. By definition of $PDel^B$, D contains at least $n - B$ processes. Let $r > 0$ and consider a $CDel$ such that $\forall j \in \Pi : \forall r' < r : CDel(r', j) = \Pi \wedge \forall r' \geq r : CDel(r, j) = D$. All processes share the same Delivered set of size at least $n - B$, thus $CDel \in PDel^B$. $CDel$ is symmetric, thus $PDel^B$ is round-symmetric. ◀

▶ **Theorem 28** (Heard-Of Characterization of $PDel^B$). *The Heard-Of predicate implemented by f_{n-B} in $PDel^B$ is $CHO_{f_{n-B}}(PDel^B) = \{cho \in (\mathbb{N}^* \times \Pi) \mapsto \mathcal{P}(\Pi) \mid \forall r > 0, \forall j \in \Pi : |cho(r, j)| \geq n - B\}$.*

Proof. We know that $Nexts_{f_{n-B}} = \{cfree(q) \mid q \in Q \wedge |\{k \in \Pi \mid \langle q.round, k\rangle \in q.received\}| \geq n - B\} = \{S \in \mathcal{P}(\Pi) \mid |S| \geq n - B\}$. We conclude by application of Theorem 20. ◀

5 Beyond Carefree Strategies: Reactionary Strategies

Sometimes, carefree strategies are not enough to capture the subtleties of a Delivered predicate. Take the one corresponding to at most F initial crashes for example: to make the most of this predicate, a strategy should remember from which processes it received a message, since it knows this process did not crash. A class of strategies which allows this is the class of reactionary strategies: they depend on messages from current and past rounds, as well as the round number. The only part of the local state these strategies cannot take into account is the set of messages received from "future" rounds, a possibility due to asynchrony.

5.1 Definition and Expressiveness Results

▶ **Definition 29** (Reactionary Strategy). Let f be a strategy, and $\forall q \in Q$, let $reac(q) \triangleq \langle q.round, \{\langle r, k\rangle \in q.received \mid r \leq q.round\}\rangle$. f is a **reactionary strategy** $\triangleq \forall q, q' \in Q : reac(q) = reac(q') \implies (q \in f \iff q' \in f)$. We write $Nexts^R_f \triangleq \{reac(q) \mid q \in f\}$ for the set of reactionary states in f. This uniquely defines f.

Even if reactionary strategies are more complex, we can still prove the same kind of results as for carefree ones, namely about validity and the existence of a reactionary strategy dominating all reactionary strategies.

▶ **Lemma 30** (Validity of Reactionary). *Let PDel be a Delivered predicate and f a reactionary strategy. Then, f is valid for PDel $\iff \forall CDel \in PDel, \forall r > 0, \forall j \in \Pi : \langle r, \{\langle r', k\rangle \mid r' \leq r \wedge k \in CDel(r', j)\} \rangle \in Nexts^R_f$.*

Proof sketch. The proof is the same as the one of Lemma 19, replacing "messages from round r" by "messages up to round r". ◀

There is also always a reactionary strategy dominating all reactionary strategies for a given Delivered predicate.

▶ **Theorem 31** (Always a Dominating Reactionary Strategy). *Let PDel a Delivered predicate and let f_{rcDom} be the reactionary strategy defined by*
$Nexts^R_{rcDom} = \{\langle r, \{\langle r', k\rangle \mid r' \leq r \wedge k \in CDel(r', j)\} \rangle \mid r > 0 \wedge CDel \in PDel\}$.
f_{rcDom} dominates all reactionary strategies for PDel.

Proof. First, f_{rcDom} is valid for *PDel* by application of Lemma 30.

As for domination, we also deduce from Lemma 30 that $\forall f$ a valid reactionary strategy for *PDel*, $Nexts^R_{f_{rcDom}} \subseteq Nexts^R_f$ and thus $f_{rcDom} \subseteq f$. Therefore, $\forall q \in Q : q \in f_{rcDom} \implies q \in f$. This gives us $runs_{f_{rcDom}}(PDel) \subseteq runs_f(PDel)$, and we conclude $PHO_{f_{rcDom}}(PDel) \subseteq PHO_f(PDel)$. We conclude that f_{rcDom} dominates all valid carefree strategies for *PDel*. ◀

5.2 Example Dominated by Reactionary Strategy

To show the usefulness of reactionary strategies, we study the Delivered predicate corresponding to reliable communication and at most F initial crashes: either processes crash initially and no message of theirs is ever delivered, or they do not and all their messages will be delivered eventually.

▶ **Definition 32** ($PDel^F_{ini}$). The Delivered predicate $PDel^F_{ini}$ for at most F initial crashes is: $\{CDel \in (\mathbb{N}^* \times \Pi) \mapsto \mathcal{P}(\Pi) \mid \exists \Sigma \subseteq \Pi : |\Sigma| \geq n - F \wedge \forall r > 0, \forall j \in \Pi : CDel(r, j) = \Sigma\}$.

As we mentioned above, it is possible to take advantage of the past by waiting in the current round for messages from processes which sent a message in a past round.

▶ **Definition 33** (Past complete strategy). The **past-complete strategy** f_{pc} is defined by $Nexts^R_{f_{pc}} = \{\langle r, [1, r] \times \Sigma\rangle \mid r > 0 \wedge \Sigma \subseteq \Pi \wedge |\Sigma| \geq n - F\}$.

▶ **Lemma 34** (f_{pc} Reactionary Dominates $PDel^F_{ini}$). *f_{pc} dominates all reactionary strategies for $PDel^F_{ini}$.*

Proof. It follows from Theorem 31, because $Nexts^R_{f_{pc}} = \{\langle r, \{\ \langle r', k\rangle \mid r' \leq r \wedge k \in \Sigma\}\rangle \mid r > 0 \wedge \Sigma \subseteq \Pi \wedge |\Sigma| \geq n - F\} = \{\langle r, \{\langle r', k\rangle \mid r' \leq r \wedge k \in CDel(r', j)\}\rangle \mid r > 0 \wedge CDel \in PDel^F_{ini}\}$. The last equality follows from the fact that Delivered sets are always the same for all Delivered collection in $PDel^F_{ini}$. ◀

Reactionary strategies can generate more complex and involved Heard-Of predicates than carefree ones. The one generated by f_{pc} for $PDel^F_{ini}$ is a good example: it ensures that all Heard-Of sets contains at least $n - F$ processes, and it also forces Heard-Of sets for a process to be non-decreasing, and for all rounds to eventually converge to the same Heard-Of set. This follows from the fact that a process can detect an absence of crash: if one message is received from a process, it will always be safe to wait for messages from this process as it did not crash and never will. Since there is no loss of message for non crashed processes, every one eventually receives a message from every process sending messages, and thus the Heard-Of sets converge.

▶ **Theorem 35** (Heard-Of Characterization of $PDel^F_{ini}$). $PHO_{f_{pc}}(PDel^F_{ini}) =$

$$\left\{ cho \in (\mathbb{N}^* \times \Pi) \mapsto \mathcal{P}(\Pi) \;\middle|\; \left(\begin{array}{ll} & \forall r > 0, \forall j \in \Pi : \left(\begin{array}{ll} & |cho(r, j)| \geq n - F \\ \wedge & cho(r, j) \subseteq cho(r + 1, j) \end{array} \right) \\ \wedge & \exists \Sigma_0 \subseteq \Pi, \exists r_0 > 0, \forall r \geq r_0, \forall j \in \Pi : cho(r, j) = \Sigma_0 \end{array} \right) \right\}$$

Proof Sketch. First, we show $\subseteq$. Let $cho \in PHO_{f_{pc}}(PDel^F_{ini})$ and t a run of f_{pc} implementing cho. By definition of f_{pc}, processes change round only when they have received at least $n - F$ messages from the current round and they have completed their past, which gives the first conjunction. Concerning the second conjunction, observe that due to fairness, there is a point in t where all messages from round 1 have been delivered. After that round (r_0), f_{pc} makes all processes wait for the messages of the living processes (Σ_0).

Then, we show $\supseteq$. Consider a run t of *cho* in iterations such that at each iteration, the delivered messages are the Heard-Of sets and the messages needed to complete the past of these sets, and every process changes round. It is a run of f_{pc} because each Heard-Of set contains at least $n - F$ processes and the past of each process is complete before playing any *next*. Moreover, there is a round after which all processes share the same Heard-Of set Σ_0 forever. Consider *CDel* such that $\forall r > 0, \forall j \in \Pi : \mathit{CDel}(r, j) = \Sigma_0$. $t \in \mathit{runs}(\mathit{CDel})$ and $\mathit{CDel} \in \mathit{PDel}^F_{ini}$. Thus, $\mathit{cho} \in PHO_{f_{pc}}(\mathit{PDel}^F_{ini})$. ◀

As for f_{pc} dominating PDel^F_{ini}, the argument is quite similar to the one for f_{n-F} dominating PDel^F, with a more subtle manipulation of Delivered collection because we need to take the past into account.

▶ **Theorem 36** (f_{pc} Dominates PDel^F_{ini}). *f_{pc} dominates PDel^F_{ini}.*

Proof Sketch. This domination proof is similar to the one for Round-Symmetric Predicates or for PDel^F. The difference is that given a Heard-Of collection in $PHO_{f_{pc}}(\mathit{PDel}^F_{ini})$, the run generating it for any valid strategy is a run in iterations where, at each iteration, all processes get delivered their Heard-Of set and the messages necessary to complete their past, and then they all change round. ◀

5.3 When The Future Serves

In the above, we considered cases where the dominating strategy is at most reactionary: only the past and present rounds are useful for generating Heard-Of collections. But messages from future rounds serve in some cases. We give an example, presenting only the intuition.

▶ **Definition 37** (PDel^1_{lost}). The Delivered predicate PDel^1_{lost} corresponding to at most 1 message lost is: $\{\mathit{CDel} \in (\mathbb{N}^* \times \Pi) \mapsto \mathcal{P}(\Pi) \mid \sum_{r>0, j \in \Pi} |\Pi \setminus \mathit{CDel}(r, j)| \leq 1\}$.

Application of our results on carefree strategy shows that the carefree strategies dominating all carefrees for this predicate is f_{n-1}. Similarly, looking at the past only allows processes to wait for $n - 1$ messages because one can always deliver all messages from the past, and then the loss might be a message from the current round. If we look at the messages from the next round, on the other hand, we can ensure that at each round, at most one message among all processes is not delivered on time.

▶ **Definition 38** (Asymmetric Strategy). Let $\mathit{after} : Q \mapsto \mathcal{P}(\Pi)$ such that $\forall q \in Q : \mathit{after}(q) = \{k \in \Pi \mid \langle q.\mathit{round} + 1, k\rangle \in q.\mathit{received}\}$, and *cfree* as in Definition 18.
Define $f_{asym} \triangleq \left\{ q \in Q \;\middle|\; \begin{array}{rl} & \mathit{cfree}(q) = \Pi \\ \vee & (|\mathit{after}(q)| = n - 1 \wedge |\mathit{cfree}(q)| = n - 1\} \end{array} \right\}$.

Intuitively, this strategy is valid because at each round and for each process, only two cases exist: either no message for this process at this round is lost, and it receives a message from each process; or one message for this process is lost at this round, and it only receives $n - 1$ messages. But all other processes receive n messages, thus change round and send their message from the next round. Since the one loss already happened, all these messages are delivered, and the original process eventually receives $n - 1$ messages from the next round.

This strategy also ensures that at most one process per round receives only $n - 1$ messages on time – the others must receive all messages. This vindicates the value of messages from future rounds for some Delivered predicates, such as the ones with asymmetry in them.

6 Related Works

Rounds Everywhere. Rounds in message-passing algorithms date at least back to their use by Arjomandi et al. [1] as a synchronous abstraction of time complexity. Since then, they are omnipresent in the literature. First, the number of rounds taken by a distributed computation is a measure of its complexity. Such round complexity was even developed into a full-fledged analogous of classical complexity theory by Fraigniaud et al. [7]. Rounds also serve as stable intervals in the dynamic network model championed by Kuhn and Osham [11]: each round corresponds to a fixed communication graph, the dynamicity following from possible changes in the graph from round to round. Finally, many fault-tolerant algorithms are structured in rounds, both synchronous [6] and asynchronous ones [3].

Although we only study message-passing models in this article, one cannot make justice to the place of rounds in distributed computing without mentioning its even more domineering place in shared-memory models. A classic example is the structure of executions underlying the algebraic topology approach pioneered by Herlihy and Shavit [9], Saks and Zaharoglou [13], and Borowsky and Gafni [2].

Abstracting the Round. Gafni [8] was the first to attempt the unification of all versions of rounds. He introduced the Round-by-Round Fault Detector abstraction, a distributed module analogous to a failure detector which outputs a set of suspected processes. In a system using RRFD, the end condition of rounds is the reception of a message from every process not suspected by the local RRFD module; communication properties are then defined as predicates on the output of RRFDs. Unfortunately, this approach does not suit our needs: RRFDs do not ensure termination of rounds, while we require it.

Next, Charron-Bost and Schiper [4] took a dual approach to Gafni's with the Heard-Of Model. Instead of specifying communication by predicates on a set of suspected processes, they used Heard-Of predicates: predicates on a collection of Heard-Of sets, one for each round r and each process j, containing every process from which j received the message sent in round r before the end of this same round. This conceptual shift brings two advantages: a purely abstract characterization of message-passing models and the assumption of infinitely many rounds, thus of round termination.

But determining which model implements a given Heard-Of predicate is an open question. As mentioned in Marić [12], the only known works addressing it, one by Hutle and Schiper [10] and the other by Drăgoi et al. [5], both limit themselves to very specific predicates and partially synchronous system models.

7 Conclusion and Perspectives

We propose a formalization for characterizing Heard-Of predicate of an asynchronous message-passing model through Delivered predicates and strategies. We also show its relevance, expressivity and power: it allows us to prove the characterizations from Charron-Bost and Schiper [4], to show the existence of characterizing predicates for large classes of strategies as well as the form of these predicates. Yet there are two aspects of this research left to discuss: applications and perspectives.

First, what can we do with this characterizing Heard-Of predicate? As mentioned above, it gives the algorithm designer a concise logical formulation of the properties on rounds a given model can generate. Therefore, it allows the design of algorithms at a higher level of abstraction, implementable on any model which can generate the corresponding Heard-Of

predicate. The characterizing predicate is also crucial to verification: it bridges the gap between the intuitive operational model and its formal counterpart. To verify a round-based algorithm for a given message-passing model, one needs only to check if its correctness for the characterizing predicate. If it is the case, we have a correct implementation by combining the algorithm with a dominating strategy; if it is not, then the algorithm will be incorrect for all predicates generated by the model.

Finally, it would be beneficial to prove more results about the existence of a dominating strategy, as well as more conditions for the dominating strategy to be in a given class. There is also space for exploring different Delivered predicates. For example, some of our results suppose that the predicate contains the total Delivered collection; what can we do without this assumption? Removing it means that faults are certain to occur, which is rarely assumed. Nonetheless, it might be interesting to study this case, both as a way to strengthen our results, and because forcing failures might be relevant when modelling highly unreliable environments such as the cloud or natural settings. Another viable direction would be to add oracles to the processes, giving them additional information about the Delivered collection, and see which Heard-Of predicates can be generated. These oracles might for example capture the intuition behind failure detectors.

References

1. Eshrat Arjomandi, Michael J. Fischer, and Nancy A. Lynch. A Difference in Efficiency Between Synchronous and Asynchronous Systems. In *Thirteenth Annual ACM Symposium on Theory of Computing*, STOC '81, pages 128–132, 1981. doi:10.1145/800076.802466.
2. Elizabeth Borowsky and Eli Gafni. Generalized FLP Impossibility Result for T-resilient Asynchronous Computations. In *Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC '93, pages 91–100. ACM, 1993. doi:10.1145/167088.167119.
3. Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The Weakest Failure Detector for Solving Consensus. *J. ACM*, 43(4):685–722, July 1996. doi:10.1145/234533.234549.
4. Bernadette Charron-Bost and André Schiper. The Heard-Of model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1):49–71, April 2009. doi:10.1007/s00446-009-0084-6.
5. Cezara Drăgoi, Thomas A. Henzinger, and Damien Zufferey. PSync: A Partially Synchronous Language for Fault-tolerant Distributed Algorithms. In *43rd Symposium on Principles of Programming Languages*, pages 400–415, 2016. doi:10.1145/2837614.2837650.
6. Michael J. Fischer and Nancy A. Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14(4):183–186, 1982. doi:10.1016/0020-0190(82)90033-3.
7. Pierre Fraigniaud, Amos Korman, and David Peleg. Towards a Complexity Theory for Local Distributed Computing. *J. ACM*, 60(5):35:1–35:26, October 2013. doi:10.1145/2499228.
8. Eli Gafni. Round-by-round Fault Detectors (Extended Abstract): Unifying Synchrony and Asynchrony. In *Seventeenth ACM Symposium on Principles of Distributed Computing*, PODC '98, pages 143–152. ACM, 1998. doi:10.1145/277697.277724.
9. Maurice Herlihy and Nir Shavit. The Topological Structure of Asynchronous Computability. *J. ACM*, 46(6):858–923, November 1999. doi:10.1145/331524.331529.
10. M. Hutle and A. Schiper. Communication Predicates: A High-Level Abstraction for Coping with Transient and Dynamic Faults. In *37th International Conference on Dependable Systems and Networks (DSN'07)*, pages 92–101, June 2007. doi:10.1109/DSN.2007.25.
11. Fabian Kuhn and Rotem Oshman. Dynamic Networks: Models and Algorithms. *SIGACT News*, 42(1):82–96, March 2011. doi:10.1145/1959045.1959064.

12 Ognjen Marić, Christoph Sprenger, and David Basin. Cutoff Bounds for Consensus Algorithms. In Rupak Majumdar and Viktor Kunčak, editors, *Computer Aided Verification*, pages 217–237. Springer International Publishing, 2017.

13 Michael Saks and Fotios Zaharoglou. Wait-Free k-Set Agreement is Impossible: The Topology of Public Knowledge. *SIAM J. Comput.*, 29(5):1449–1483, March 2000. `doi:10.1137/S0097539796307698`.

14 Adam Shimi, Aurélie Hurault, and Philippe Quéinnec. Characterizing Asynchronous Message-Passing Models Through Rounds. *CoRR*, 2018. `arXiv:1805.01657`.

You Only Live Multiple Times: A Blackbox Solution for Reusing Crash-Stop Algorithms In Realistic Crash-Recovery Settings

David Kozhaya
ABB Corporate Research, Switzerland
david.kozhaya@ch.abb.com

Ognjen Marić
Digital Asset, Switzerland
ogi.yolmt@mynosefroze.com

Yvonne-Anne Pignolet
ABB Corporate Research, Switzerland
yvonne-anne.pignolet@ch.abb.com

Abstract

Distributed agreement-based algorithms are often specified in a crash-stop asynchronous model augmented by Chandra and Toueg's unreliable failure detectors. In such models, correct nodes stay up forever, incorrect nodes eventually crash and remain down forever, and failure detectors behave correctly forever eventually, However, in reality, nodes as well as communication links both crash and recover without deterministic guarantees to remain in some state forever.

In this paper, we capture this realistic temporary and probabilitic behaviour in a simple new system model. Moreover, we identify a large algorithm class for which we devise a property-preserving transformation. Using this transformation, many algorithms written for the asynchronous crash-stop model run correctly and unchanged in real systems.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Crash recovery, consensus, asynchrony

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.19

Related Version A full version is available at `https://arxiv.org/abs/1811.05007`.

1 Introduction

Distributed systems comprise multiple software and hardware components that are bound to eventually fail [10]. Such failures can cause service malfunction or unavailability, incurring significant costs to mission-critical systems, e.g., automation systems and on-line transactions. The failures' impact can be minimized by protocols that let systems agree on actions despite failures. As a consequence, many variants of the agreement or the consensus problem [29] under different assumptions have been studied. Of particular importance are synchrony and failure model assumptions, as they determine the problem's complexity.

In the simplest failure model, often called the *crash-stop* model, a process fails by stopping to execute its protocol and never recovers. Combining this failure model with an asynchronous system, i.e., a system without bounds on execution delays or message latency, makes it impossible to distinguish a crashed from a very slow process. This renders consensus-like problems unsolvable deterministically [16], already in this very simple failure model. To circumvent this impossibility, previous works have investigated ways to relax the underlying

© David Kozhaya, Ognjen Maric, and Yvonne-Anne Pignolet;
licensed under Creative Commons License CC-BY
22nd International Conference on Principles of Distributed Systems (OPODIS 2018).
Editors: Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira; Article No. 19; pp. 19:1–19:17

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

asynchrony assumption either explicitly, e.g., by using partial synchrony [12], or implicitly, by defining oracles that encapsulate time, e.g., failure detectors [8]. The result is a large and rich body of literature that builds on top of the former and latter techniques to solve consensus-like problems in the presence of crash-stop failures. Typically, the respective proofs rely on assumptions of the "eventually forever" form: the correct nodes stay up forever, incorrect nodes eventually crash and remain down forever, and failure detectors produce wrong output in the beginning, but provide correct results forever eventually.

However, such "eventually forever" assumptions are not met by real distributed systems. In reality, processes may crash but their processors reboot, and the recovered process rejoins the computation. Communication might also fail at any point in time, but get restored later. Hence, the failure and recovery modes of processes as well as communication links are in reality probabilistic and temporary [13, 15, 31], especially in systems incorporating many unreliable off-the-shelf low-cost devices and communication technologies. This led to the development of *crash-recovery* models, where processes repeatedly leave and join the computation unannounced. This requires new failure detector definitions and new consensus algorithms built on top of these failure detectors [1, 24, 11, 21] as well as completely new solutions (without failure detectors) that consider different classes of failures, namely classified according to how many times a process can crash and recover [25]. However, such solutions eliminate the "eventually forever" assumptions only on the processes' level and not for the communication and failure detectors. Moreover, these works derive new algorithms tailored to crash-recovery settings. This leaves unanswered the question: can the plethora of existing crash-stop algorithms be reused (ideally, unchanged) in crash-recovery settings?

To this end, this paper investigates how to re-use consensus algorithms defined for the crash-stop model with reliable links and failure detectors in a more realistic crash-recovery model, where processes and links can crash and recover probabilistically and for an unbounded number of times. Our models allow *unstable* nodes, i.e., nodes that fail and recover infinitely often. These are often excluded or limited in number in other models. In contrast, we explicitly allow unstable behavior of any number of processes and links, by modeling communication problems and crash-recovery behaviors as probabilistic and temporary, rather than deterministic and perpetual. Our system model, similar to existing models that rely on probabilistic factors, e.g., coin flips, comes with the trade-off of solving consensus (namely the termination property), with probability 1, rather than deterministically.

However, unlike existing solutions that incorporate probabilistic behavior, our approach does not aim at inventing new consensus algorithms but rather focuses on using existing deterministic ones to solve consensus with probability 1. Our approach is modular: we build a wrapper that interacts with a crash-stop algorithm as a black box, exchanges messages with other wrappers and transforms these messages into messages that the crash-stop algorithm understands. We then formally define classes of algorithms and safety properties for which we prove that our wrapper constructs a system that preserves these properties. Additionally, we show that termination with probability 1 is guaranteed for wrapped algorithms of this class. Moreover, this class is wide and includes the celebrated Chandra-Toueg algorithm [8] as well as the instantiation of the indulgent framework with failure detectors from [20] . Our work allows such algorithms to be ported unchanged to our crash-recovery model. Hence applications built on top of such algorithms can run in real systems with crash-recovery behavior by simply using our wrapper.

Contributions: To summarize, our main contributions are:

- New system models that capture probabilistic and temporary failures and recoveries of processes and communication links in real distributed systems (described in Section 3)

- A wrapper framework that allows a wide class of crash-stop consensus algorithms to be used unchanged in our more realistic models (described in Section 4)
- Formal properties describing which crash-stop consensus algorithms benefit from our framework and hence can be reused to solve consensus in crash-recovery settings (described in Sections 5 and 6)

In addition to the sections presenting our contributions, we discuss related work in Section 2 and conclude the paper in Section 7. Due to space limitations, we defer most proofs and some formalization details to the full version, available at `https://arxiv.org/abs/1811.05007`.

2 Related Work

Several works addressed the impossibility of asynchronous consensus. One direction exploits the concept of partial synchrony [12], in which an asynchronous system becomes synchronous after some unknown global stabilization time (GST) for a bounded number of rounds. For the same model, ASAP [3] is a consensus algorithm where every process decides no later than round $GST + f + 2$ (optimal). Another direction augments asynchronous systems with failure detector oracles, and builds asynchronous consensus algorithms on top [8]. These detectors typically behave erratically at first, but eventually start behaving correctly forever. Like with partial synchrony, the intuitive expectation is that failure detectors must only behave correctly for "sufficiently long" instead of forever [8]; however, quantifying "sufficiently long" is impossible in a purely asynchronous model [9]. Both lines of work initially investigated crash-stop failures of processes. In real systems processes as well as network links crash and recover multiple times and sometimes even indefinitely. This gave rise to a large body of literature that studied how to adapt the two lines of work to crash-recovery behavior of processes and links. We next survey some of this literature.

Failure detectors and consensus algorithms for crash recovery: Dolev et al. [11] consider an asynchronous environment where communication links first lose messages arbitrarily, but eventually communication stabilizes such that a majority of processes forms a forever strongly connected component. Processes belonging to such a strongly-connected component are termed correct, and the others faulty. Process state is always (fully) persisted in stable storage. The authors propose a failure detector that allows the correct processes to reach a consensus decision and show that the rotating coordinator algorithm [8] works unchanged in their setting, as long as all messages are constantly retransmitted. This relies on piggybacking all previous messages onto the last message, and regularly retransmitting the last message. As this yields very large messages, they also propose a modification of [8] for which no piggybacking is necessary. While our results also rely on strongly connected components, we do not require their existence to be deterministic nor perpetual. We also do not require piggybacking in order for algorithms like [8] to be used unchanged.

Oliveira et al. [28] consider a crash-recovery setting with correct processes that may crash only finitely many times (and thus eventually stay up forever) and faulty processes that permanently crash or crash infinitely often. As in [8], the authors note that correct processes only need to stay up for long enough periods in practice (rather than forever), but this cannot be expressed in the asynchronous model. The authors take the consensus algorithm of [30] which uses stubborn links and transform it to work in the crash-recovery setting by logging every step into stable storage and adding a fast-forward mechanism for skipping rounds. Hurfin et al. [22] describe an algorithm using the $\diamond\mathcal{S}$ detector in the crash-recovery case. The notions of correct/faulty processes and of failure detectors are the same as in Oliviera

et al [28]. Their algorithm is however more efficient when using stable storage compared to [28]: there is only one write per round (of multiple data), and the required network buffer capacity for each channel (connecting a pair of processes) is one. Compared to [28] and [22] our system does not regard processes that crash and recover infinitely often as faulty and hence we allow such "unstable" processes to implement consensus.

Aguilera et al. [1] consider a crash-recovery system with lossy links. They show that previously proposed failure detectors for the crash-recovery setting have anomalous behaviors even in synchronous systems when considering unstable processes, i.e., processes that crash and recover infinitely often. The authors propose new failure detectors to mitigate this drawback. They also determine the necessary conditions regarding stable storage that allow consensus to be solved in the crash-recovery model, and provide two efficient consensus algorithms: one with, and one without using stable storage. Unlike [1], we do not exclude unstable processes from implementing consensus, thus our model tolerates a wider variety of node behavior. Furthermore, our wrapper requires no modifications to the existing crash-stop consensus algorithms, as it treats them as black-boxes.

Modular Crash-Recovery Approaches: Similar to [1], Freiling et al. [18] investigate the solvability of consensus in the crash-recovery model under varying assumptions, regarding the number of unstable and correct processes and what is persisted in stable storage. They reuse existing algorithms from the crash-stop model in a modular way (without changing them) or semi-modular way, with some modifications to the algorithm (as in the case of [8]). Similar to our work, they provide algorithms to emulate a crash-stop system on top of a crash-recovery system. Our work, however, always reuses algorithms in a fully modular way, and we define a wide class of algorithms for which such reuse is possible. Furthermore, as we model message losses, processes crashes, and process recoveries probabilistically, our results also apply if processes are unstable, i.e., crash and recover infinitely often.

Randomized Consensus Algorithms: Besides the literature that studied deterministic consensus algorithms, existing works have also explored randomized algorithms to solve "consensus with probability 1". These include, for example, techniques based on using random *coin-flips* [2, 5, 17] or *probabilistic schedulers* [7]. In systems with dynamic communication failures, multiple randomized algorithms [27, 26] addressed the *k-consensus* problem, which requires only k processes to eventually decide. Moniz et al. [27] considered a system with correct processes and a bound on the number of faulty transmission. In a wireless setting, where multiple processes share a communication channel, Moniz et al. [26] devise an algorithm tolerating up to f Byzantine processes and requires a bound on the number of omission faults affecting correct processes. In comparison, our work in this paper does not use randomization in the algorithm itself: we focus on using existing deterministic algorithms to solve consensus (with probability 1) in networks with probabilistic failure and recovery patterns of processes and links.

3 System Models

We start by defining the notation we use, and then define general concepts common to all of our models. Then, we define each of our models in turn.

Notation: Given a set S, we define $S_\perp$ to be the set $S \cup \{\perp\}$, where $\perp$ is a distinguished element not present in S. The set of finite sequences over a set S is denoted by S^*. We also

call sequences *words*, when customary. Given a non-empty sequence, *head* defines its first element, *tail* the remainder of the sequence, and, if the sequence is finite, *last* its last element. Given two sequences u and v, where u is finite, $u \cdot v$ denotes their concatenation. For a word u, $|u|$ denotes the length of u. Letting $u(i)$ be the i-th letter of u, we say that u is a *subword* of v if there exists a strictly monotone function $f : \mathbb{N} \to \mathbb{N}$ such that $u(i) = v(f(i))$, for $1 \leq i \leq |u|$ if u is finite, and for all $i \in \{1, 2, \ldots\}$ if u is infinite. Analogously, v is a *superword* of u.

We denote the space of partial functions (maps) between sets A and B by $A \rightharpoonup B$. Note that $(A \to B) \subseteq (A \rightharpoonup B)$.

Common concepts: We consider a fixed finite set of processes $\Pi = \{1 \ldots N\}$, and a fixed countable set of values, denoted $\mathcal{V}$. For each algorithm there is an algorithm-specific countable set of local states Σ_p, for each process $p \in \Pi$. For simplicity, we restrict ourselves to algorithms where $\Sigma_p = \Sigma_q$, $\forall p, q \in \Pi$. Note that this does not exclude algorithms that take decisions based on identifiers. We define the global state space $\Sigma = \prod_{p \in \Pi} \Sigma_p$. Given a $s \in \Sigma$, we define $s_p \in \Sigma_p$ as the projection of s to its p-th component.

A *property* over an alphabet A is a set of infinite words over A. We use standard definitions of liveness and safety properties [4]. A property P is a *safety property* if, for every infinite word $w \notin P$, there exists a finite prefix u of w such that the concatenation $u \cdot v \notin P$ for all infinite words v. Intuitively, the prefix u is "bad" and not recoverable from. A property P is a *liveness property* if for any finite word u there exists an infinite word v such that $u \cdot v \in P$. Intuitively, "good" things can always happen later.

In this paper, we are interested in preserving properties over the alphabet Σ between the crash-stop and crash-recovery versions of an algorithm. In particular, we assume that the local states Σ_p are records, with two distinguished fields: *inp* of type $\mathcal{V}$ and *dec* of type $\mathcal{V}_\perp$. Intuitively, a *dec* value of $\perp$ indicates that the process has not decided yet. For an infinite word w over the alphabet Σ, let $w(i, p)$ denote the local state of the process p at the i-th letter of the word. Let us state the standard safety properties of consensus in our notation.

Validity. Decided values must come from the set of input values. Formally, validity describes the set of words w such that $\forall p,\ i,\ v.\ w(i,p).dec = v \wedge v \neq \perp \implies \exists q.\ w(1,q).inp = v$

Integrity. Processes do not change their decisions. Formally, integrity describes the set of words w such that $\forall p,\ i.\ v.\ w(i,p).dec = v \wedge v \neq \perp \implies \forall i' > i.\ w(i',p).dec = v$

(Uniform) Agreement. No two processes ever make different non-$\perp$ decisions. Formally, $\forall p,\ q,\ i,\ j.\ w(i,p).dec \neq \perp \neq w(j,q).dec \implies w(i,p).dec = w(j,q).dec$

To simplify our preservation results for safety properties, our models store information about process failures separately from Σ. As a consequence, the standard crash-stop termination property cannot be expressed as a property over Σ: it is conditioned on a process not failing. However, we do not directly use the crash-stop notion of termination and we omit this definition here. Instead, we will prove the following property for the algorithms in our probabilistic crash-recovery model:

Probabilistic crash-recovery termination. With probability 1, all processes eventually decide.

3.1 The crash-stop model

Our definition of the crash-stop model is standard and closely follows [8]. We assume an asynchronous environment, with processes taking steps in an interleaved fashion. Processes communicate using reliable links, and can query failure detectors.

Failure detectors: A *failure pattern* fp is an infinite word over the alphabet 2^{Π}. Intuitively, each letter is the set of failed processes in a transition step of a run of a transition system. A *failure detector* with range $\mathcal{R}$ is a function from failure patterns to properties over the alphabet $\mathcal{R}$.[1] A failure detector D is *unreliable* if $D(fp)$ is a liveness property for all fp. Intuitively, a detector constrains how the failure detector outputs (the $\mathcal{R}$ values) must depend on the failure pattern of a run, and unreliable detectors can produce arbitrary outputs in the beginning. We write $FD(\mathcal{R})$ for the set of all detectors with range $\mathcal{R}$.

Algorithms and algorithm steps: The type of *crash-stop steps* over a message space $\mathcal{M}$ and a failure detector range $\mathcal{R}$, written $\mathit{CSS}(\mathcal{M}, \mathcal{R})$ is defined as a pair of functions of types:

$\mathit{next} : \Sigma_p \times (\Pi \times \mathcal{M})_\perp \times \mathcal{R} \to \Sigma_p, \quad \mathit{send} : \Sigma_p \to (\Pi \rightharpoonup \mathcal{M}).$

Intuitively, given zero or one messages received from some other processes and an output of the failure detector, a step maps the current process state to a new state, and maps the new state to a set of messages to be sent, with zero or one messages sent to each process.

A *crash-stop algorithm* $\mathcal{A}$ over Σ, $\mathcal{M}$ and $\mathcal{R}$ is a tuple $(I, \mathit{step}, D, N_f)$ where:

(1) $I \subseteq \Sigma$ is the *finite* set of initial states,
(2) $\mathit{step} \in \mathit{CSS}(\mathcal{M}, \mathcal{R})$ is the step function,
(3) $D \in FD(\mathcal{R})$ is a failure detector, and
(4) $N_f < N$ is the resilience condition, i.e., the number of failures tolerated by the algorithm (recall that we consider a fixed N).

We refer to the components of an algorithm $\mathcal{A}$ by $\mathcal{A}.I$, $\mathcal{A}.\mathit{step}$, $\mathcal{A}.D$ and $\mathcal{A}.N_f$.

Configurations: As noted earlier, we focus on preserving properties over Σ between crash-stop and crash-recovery models. However, Σ contains insufficient information to model the algorithm's crash-stop executions (runs). In particular, to account for

(1) asynchronous message delivery and
(2) process failures,

we must extend states to *configurations*. A *crash-stop configuration* is a triple (s, M, F) where: $s \in \Sigma$ is the (global) state, $M \subseteq \Pi \times \Pi \times \mathcal{M}$ is the set of *in-flight messages*, where $(p, q, m) \in M$ represents a message m that was sent to p by q, and $F \subseteq \Pi$ is the set of failed processes. As with algorithms, we refer to the components of a configuration c by $c.s$, $c.M$ and $c.F$.

Step labels and transitions: While the algorithm steps are deterministic, the asynchronous transition system is not: any (non-failed) process can take a step at any point in time, with different possible received messages, and different failure detector outputs. Accessing this non-determinism information is useful in proofs, so we extract it as follows. A *crash-stop step label* is a quadruple $(p, \mathit{rmsg}, \mathit{fails}, \mathit{fdo})$, where:

(1) $p \in \Pi$ is the process taking the step,
(2) $\mathit{rmsg} \in (\Pi \times \mathcal{M})_\perp$ is the message p receives in the step ($\perp$ modeling a missing message),
(3) $\mathit{fails} \subseteq \Pi$ is the set of processes failed at the end of the step, and
(4) $\mathit{fdo} \in \mathcal{R}$ is p's output of the failure detector.

A *crash-stop step* of the algorithm $\mathcal{A}$ is a triple (c, l, c'), where c and c' are configurations and l is a label. Crash-stop steps must satisfy the following properties: (i) p is not failed at the start of the step. (ii) p takes a step according to the label and $\mathcal{A}$'s rules, and the other

[1] This definition does not distinguish which process received the output, which is sufficient for $\diamond\mathcal{S}$. The definition can be easily extended to other failure detectors like $\diamond\mathcal{W}$.

processes do not move. (iii) If a message was received, then it was in flight; the received message is removed from the set of in-flight messages, while the produced messages are added. (iv) Failed processes do not recover, i.e., $c.F \subseteq c'.F = \mathit{fails}$.

Algorithm runs: A finite (respectively infinite) *crash-stop run* of $\mathcal{A}$ is a finite (infinite) alternating sequence c_0, l_0, $c_1 \ldots$ of configurations and labels, that ends in a configuration if finite, such that the initial state is allowed by the algorithm, (c_i, l_i, c_{i+1}) are valid steps, and the resilience condition is satisfied. Furthermore, the output of the failure detector must satisfy the condition of the failure detector. Such a run has *reliable links* if all in-flight messages eventually get delivered, unless the sender or the receiver is faulty. The *crash-stop system* of the algorithm $\mathcal{A}$ is the sequence of all crash-stop runs of $\mathcal{A}$. The *crash-stop system with reliable links* of the algorithm $\mathcal{A}$ is the set of all crash-stop runs with reliable links.

As mentioned before, we are interested in properties that are sequences of global states. In this sense, runs contain too much information (e.g., in-flight messages). Thus, given a run ρ, we define its *state trace* $tr_s(\rho)$, obtained by removing the labels and projecting configurations onto just the states. We introduce a notion of a *state property*: an infinite sequence of (global) states. The crash-stop system (with or without reliable links) satisfies a state property P if for every run ρ of the system, $tr_s(\rho) \in P$. We later show that our crash-recovery wrappers for crash-stop protocols preserve important state properties of crash-stop algorithms. Lastly, we note down a simple property of crash-stop runs.

▶ **Lemma 1** (Reliable links irrelevant for prefixes). *Let ρ be a finite crash-stop run of $\mathcal{A}$. Then, ρ can be extended to an infinite crash-stop run of $\mathcal{A}$ that has reliable links (intuitively, by eventually delivering all in-flight messages, M).*

Summary of time and failure assumptions

Time. Processes are asynchronous and have no notion of time. Links are asynchronous.

Failures. Processes can fail by halting forever, while links interconnecting them do not fail.

3.2 The lossy synchronous crash-recovery model

We next define our first crash-recovery model. Formally, this a lossy synchronous crash-recovery model, with non-deterministic, but not probabilistic losses, crashes, and recoveries. We use it to prove the preservation of safety properties without taking probabilities into account, since they are not used in such arguments. In this model, we will not distinguish between volatile and persistent memory of a process. Instead, we assume that all memory is persistent. This can be emulated in practice by persisting all volatile memory before taking any actions with side-effects (such as sending network messages). Finally, while the model is formally synchronous, in that all processes take steps simultaneously, it also captures processing delays, as a slow process behaves like a process that crashes and later recovers.

Algorithms and algorithm steps: A *crash-recovery step* over a message space $\mathcal{M}$, written $\mathit{CRS}(\mathcal{M})$ is defined as the pair of functions *next* and *send*, with types:

$\mathit{next} : \Sigma_p \times (\Pi \rightharpoonup \mathcal{M}) \to \Sigma_p \quad \mathit{send} : \Sigma_p \to (\Pi \to \mathcal{M})$.

In other words, a step determines the new state based on the current state and the map of received messages. Given the new state, a process sends a message to every other processes (including itself). Compared to the asynchronous setting (Section 3.1), in this model:

1. A process can receive multiple messages simultaneously (rather than receiving at most one message in a step).
2. Every process sends a message to every other process at each step. The wrapped algorithms in later sections satisfy this by sending heartbeat messages, if there is nothing else to exchange. Delivery of sent messages is not guaranteed in this model.
3. No failure detector oracle is specified. The synchrony assumption of this model inherently provides spurious failure detection: each process can suspect all peers it did not hear from in the last message exchange. This is in fact exactly what we will use to provide failure detector outputs to the "wrapped" crash-stop algorithms run in this setting[2].

A *crash-recovery algorithm* $\mathcal{A}^R$ over Σ, $\mathcal{M}$ is a pair (I, step) where: $I \subseteq \Sigma$ is a finite set of initial states, and $\mathit{step} \in \mathit{CRS}(\mathcal{M})$ is the step function.

Configurations: As in the crash-stop case, we require more than just the global states to model algorithm executions; we hence introduce configurations. These, however, differ from those for the crash-stop setting. As communication is synchronous, we need not store the in-flight messages; they are either delivered by the end of a step, or they are gone. Furthermore, as processes take steps synchronously, we can introduce a global step number. [3]

A *crash-recovery configuration* is a tuple (n, s, F) where $n \in \mathbb{N}$ is the step number, $s \in \Sigma$ is the (global) state, $F \subseteq \Pi$ is the set of failed processes. We denote the set of all crash-recovery configurations by $\mathcal{C}^R$. Note that this set is countable. This will allow us to impose a Markov chain structure on the system in the later model.

Step labels and transitions: As in the crash-stop setting, we use labels to capture all sources of non-determinism in a step. We will use these labels to assign probabilities to different state transitions in the probabilistic model of the next section.

A *crash-recovery step label* is a pair $(\mathit{rmsgs}, \mathit{fails})$, where:

- $\mathit{rmsgs} : \Pi \to (\Pi \rightharpoonup \mathcal{M})$ denotes the message received in the step; $\mathit{rmsgs}(p)(q)$ is the message received by p on the channel from q to p. As we assume that q always attempts to send a message to p, if $\mathit{rmsgs}(p)(q)$ is undefined ($\mathit{rmsgs}(p)$ is a partial function), then either the message on this channel was lost in the step, or the sender q has failed.
- $\mathit{fails} \subseteq \Pi$ is the set of processes that are failed at the end of the step.

The *crash-recovery steps* (or transitions) of $\mathcal{A}$, written $\mathit{Tr}(\mathcal{A})$, is the set of all triples (c, l, c'), where (i) only processes that are up handle their messages (ii) messages from failed senders are not received (iii) failed processes $\mathit{fails} = c'.F$.

Algorithm runs: A finite, resp. infinite *crash-recovery run* of $\mathcal{A}$ is a finite, resp. infinite alternating sequence c_0, l_0, $c_1 \ldots$ of (crash-recovery) configurations and labels, ending in a configuration if finite, such that: $c_0.s \in \mathcal{A}.I$, i.e., the initial state is allowed by the algorithm, and $(c_i, l_i, c_{i+1}) \in \mathit{Tr}(\mathcal{A})$ for all i, that is, each step is a valid crash-recovery transition of $\mathcal{A}$. The *crash-recovery system* of algorithm $\mathcal{A}$ is the sequence of all crash-recovery runs of $\mathcal{A}$.

---

[2] Similar to Gafni's round-by-round fault detectors [19]; in our case, the detectors are "step-by-step"

[3] We use global step numbers later in the probabilistic model, to assign failure probabilities for processes and links (e.g., a probability $p_{ij}(t)$ of a message from i to j getting through, if sent in the t-th step).

Summary of time and failure assumptions

Time. Processes are synchronous and operate in a time-triggered fashion. Links are synchronous (all delivered messages respect a timing upper bound on delivery).

Failures. Processes can fail and recover infinitely often. In every time-step, a link can be either crashed or correct. A crashed link drop the messages (if any) sent over it.

3.3 The probabilistic crash-recovery model

We now extend the lossy synchronous crash-recovery model to a probabilistic model, where both the successful delivery of messages and failures follow a distribution that can vary with time. A *probabilistic network* $\mathcal{N}$ is a function of type $\Pi \times \Pi \times \mathbb{N} \rightarrow [0,1]$, such that $\exists \epsilon_{\mathcal{N}} > 0.\ \forall p,\ q,\ t.\ \mathcal{N}(q,p,t) > \epsilon_{\mathcal{N}}$ Intuitively, $\mathcal{N}(q,p,t)$ is the delivery probability for a message sent from p to q at time (step number) t. A *probabilistic failure pattern* $F^{\mathcal{P}}$ is a function $\Pi \times \mathbb{N} \rightarrow [0,1]$, such that $\exists \epsilon_F > 0.\ \forall p,\ t.\ \epsilon_F < F^{\mathcal{P}}(p,t) < 1 - \epsilon_F$. Intuitively, $F^{\mathcal{P}}(p,t)$ gives the probability of p being up at time t.[4]

Given a crash-recovery algorithm $\mathcal{A}^R$, a probabilistic network $\mathcal{N}$ and failure pattern $F^{\mathcal{P}}$, a *probabilistic crash-recovery system* $\mathcal{S}^R(\mathcal{A}^R, \mathcal{N}, F^{\mathcal{P}})$ is the Markov chain [6] with:

- The set of states $\mathcal{C}^R$, i.e., the crash-recovery configuration set.
- The transition probabilities $P(c)(c')$, defined as follows. Intuitively, a transition from c to c' is only possible if it is possible in the lossy synchronous crash-recovery model. The probability of this transition is calculated by summing over all labels that lead from c to c', and giving each such label a weight. Hence, we define $P(c) = \mathit{nf}_{\mathit{trans}}(c) \cdot \mathit{trans}(c)$, where $\mathit{nf}_{\mathit{trans}}(c)$ is the normalization factor for $\mathit{trans}(c)$ and $\mathit{trans}(c)(c')$ is defined as

$$\begin{aligned}\mathit{trans}(c)(c') = \sum_{\mathit{rmsgs},\mathit{fails}} & [\![(c, (\mathit{rmsgs}, \mathit{fails}), c') \in \mathit{Tr}(\mathcal{A}^R)]\!] \cdot \prod_{p,q} ([\![\mathit{rmsgs}(p)(q) \text{ defined}]\!] \cdot \mathcal{N}(q,p,c.n) \\ & + [\![\mathit{rmsgs}(p)(q) \text{ undefined}]\!] \cdot [\![q \notin c.F]\!] \cdot (1 - \mathcal{N}(q,p,c.n))) \\ & \cdot \prod_{p} ((1 - F^{\mathcal{P}}(p,\ c'.n)) \cdot [\![p \in \mathit{fails}]\!] + F^{\mathcal{P}}(p,\ c'.n) \cdot [\![p \notin \mathit{fails}]\!]).\end{aligned}$$

  Here, $[\![\cdot]\!]$ maps the Boolean values true and false to 1 and 0 respectively. Note that the only non-determinism in the transitions of $\mathit{Tr}(\mathcal{A}^R)$ comes exactly from the behavior we deem probabilistic: messages being dropped by the network, and process failures.
  It is easy to see that $\mathit{nf}_{\mathit{trans}}(c)$ is well defined for all c, as for a fixed configuration c, $\mathit{trans}(c)(c')$ is non-zero for only finitely many configurations c'.
- The distribution over the initial states defined by $\iota = \mathit{norm}(\mathit{init})$, where
  - $P_f(F) = \prod_{p \notin F} F^{\mathcal{P}}(p,0) \cdot \prod_{p \in F} (1 - F^{\mathcal{P}}(p,0))$,
  - $\mathit{init}(n,s,F) = P_f(F) \cdot [\![n = 0]\!] \cdot [\![s \in \mathcal{A}^R.I]\!]$,

  and norm normalizes the probabilities. Note that normalization is possible, since we assumed that after fixing N, each algorithm comes with a finite set of initial states.

Summary of time and failure assumptions

Time. Processes are synchronous and operate in a time-triggered fashion. Links are synchronous (all delivered messages respect a timing upper bound on delivery).

[4] Considering infinite time, the upper and lower bounds on $F^{\mathcal{P}}(p,t)$ ensure that, with probability 1, there is a time when process p is up.

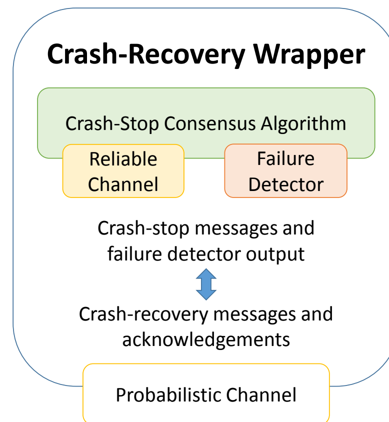


Figure 1 Wrapper concept.

Failures. Processes and links can fail and recover infinitely often. At the beginning of any time-step a crashed process/link can recover with positive probability and a correct process/link can fail with positive probability.

Important: Results in Sections 4 and 5 hold for both crash-recovery models (Section 3.2 and 3.3). Results in Section 6 rely on probabilities.

4 Wrapper for Crash-Stop Algorithms

We now define the transformation of a crash-stop algorithm $\mathcal{A}$ into a crash-recovery algorithm $\mathcal{A}^R$. Intuitively, we do this by (also illustrated in Figure 1):

- Generating a synchronous crash-recovery step using a series of crash-stop steps. Each step in the series handles one individual received message, allowing us to iteratively handle multiple simultaneously incoming messages and bridge the synchrony mismatch between the crash-stop and crash-recovery models.
- Using round-by-round failure detectors to produce the failure detector outputs to be fed to the crash-stop algorithm. These outputs are from the set 2^{Π}.[5]
- Providing reliable links, as required by the crash-stop algorithm. During each crash-recovery step, we buffer all outgoing messages of a process, and send them repeatedly in the subsequent crash-recovery steps, until an acknowledgment is received.

We first define the message and state spaces of the crash-recovery version $\mathcal{A}^R$ of a given crash-stop algorithm $\mathcal{A}$ as follows:

- In $\mathcal{A}^R$, we send a pair of messages to each process in each step:
  (1) the actual *payload* message (from $\mathcal{A}$), replaced by a special *heartbeat* message hb being sent when no payload needs to be sent, and
  (2) an *acknowledgment* message, confirming the receipt of the last message on the channel in the opposite direction,

[5] We could instead produce outputs that never suspect anyone, since no process crashes forever in our probabilistic model. However for a weaker model that we define in the full version (where processes are allowed to crash forever), we need failure detectors that suspect processes.

- The local state s_p of a process p has three components (st, $buff$, $acks$): (1) st stores the state of p in the target crash-stop algorithm; (2) $buff$ represents p's outgoing message buffers, with one buffer for each process (including one for p); and (3) $acks(q)$ records the last message that p received from q. The buffers are LIFO, a choice which proves crucial for our termination proof (Section 6).

Next, given a crash-recovery state s, a process p, and the messages $rmsgs$ received by p in the given round, we define $unfold(p, s, rmsgs)$, p's local step unfolding for s and $rmsgs$. We define $unfold(p, s, rmsgs)$ as the sequence of intermediate steps p takes. Said differently, $unfold(p, s, rmsgs)$ is a sequence of crash-recovery states and crash-stop labels $s_0, l_0, s_1, l_1, s_2, \ldots s_n$, where the intermediate state s_i represents the state of p after processing the message of the i-th process. In a crash-recovery run, p transitions directly from s_0 to s_n. The intermediate states are listed here separately to intuitively show how s_n is computed. The unfolding also allows to relate traces of $\mathcal{A}$ and $\mathcal{A}^R$ more easily in our proofs, as we produce a crash-stop run from a crash-recovery run when proving properties of the wrapper. The content of p's buffers changes as we progress through the states s_i of $unfold(p, s, rmsgs)$, as the wrapper routes the messages to $\mathcal{A}$ and receives new ones from it. The failure detector output (recorded in the labels l_i) remains constant through the unfolding: all processes from whom no message was received in the crash-recovery step are suspected. Finally, the set of failed processes in each label is defined to be empty. We emulate the process recovery that is possible in the crash-recovery model by crash-stop runs in which no processes fail.

Finally, given a crash-stop algorithm $\mathcal{A}$ with an unreliable failure detector and with the per-process state space Σ_p, we define its *crash-recovery version* $\mathcal{A}^R$ where:

- the initial states of $\mathcal{A}^R$ constitute the set of crash-recovery configurations c such that there exists a crash-stop configuration $c_s \in \mathcal{A}.I$ satisfying the following: (i) the initial states of c and c_s correspond to each other and in c all buffers are empty, (ii) no messages are acknowledged, and (iii) the failed processes in c and c_s are the same.
- the next state of a process p is computed by unfolding, based on the messages p received in this round.
- the message that p sends to a process q pairs the first element of p's (LIFO) buffer for q with the acknowledgment for the last message that p received from q.
- the execution is short-circuited as soon as a process decides. This is achieved by broadcasting a message to all other processes, announcing that it has decided. When processes receives such a message, they immediately decide and short-circuits their execution.

Short-circuiting behavior is a common pattern for consensus algorithms [8, 20]. It can be applied in a black-box way and it is sound for any crash-stop consensus algorithm.

A more formal description of the wrapper can be found in the full version. We overload the function symbol $tr_s(\cdot)$ to work on both crash-stop and crash-recovery runs. Given a crash-recovery run ρ^R, we define its *state trace*, $tr_s(\rho^R)$, as the sequence obtained by first removing the labels, then projecting each resulting configuration c onto $c.s$, and finally projecting each local state $c.s_p$ onto $c.s_p.st$. Note that both crash-stop and crash-recovery state properties are sequences of states from the same state space Σ.

5 Preservation Results

As our first main result, we show that crash-recovery versions of algorithms produced by our wrapper preserve a wide class of safety properties. The class includes the safety properties of consensus: validity, integrity and agreement (Section 3). In other words, if a trace of a crash-recovery version of an algorithm violates a property, then some crash-stop trace

of the same algorithm also violates that property. We show this in the non-probabilistic crash-recovery model. However, the result also translates to the probabilistic model, since all allowed traces of the probabilistic model are also traces of the non-probabilistic one.

Preserving all safety properties for all algorithms and failure detectors would be too strong of a requirement, for two reasons. First, as our crash-recovery model assumes nothing about link or process reliability, in finite runs we can give no guarantees about the accuracy of the simulated failure detectors. Second, the crash-recovery model is synchronous, meaning that different processes take steps simultaneously. This is impossible in the crash-stop model, which is asynchronous. Thus, the following simple safety property *OneChanges* defined by "the local state of at most one process changes between two successive states in a trace" holds in the crash-stop model, but not in the crash-recovery model (equivalently, we can find a crash-recovery trace, but not a crash-stop trace that violates the property).

We work around the first problem by assuming that the crash-stop algorithms use unreliable failure detectors. For the second problem, we restrict the class of safety properties that we wish to preserve as follows. Consider a property P. Let $u \notin P$ and w be runs with the same initial states (i.e., $u(1) = w(1)$) such that u is a subword of w (recall we define subwords earlier). P then belongs to the class of properties *not repairable through detours* if $w \notin P$ for all such u and w. Intuitively, this means that the sequence of states represented by u inherently violates P; so adding "detours" by the means of additional intermediate states (forming w) does not help satisfy P.

The property *OneChanges* is an example of a safety property that *is* repairable through detours: we can take *any* word that violates *OneChanges* and extend it to a word that does not violate *OneChanges*. However, we can easily show that the following safety properties are not repairable through detours:

- The safety properties of consensus. E.g., consider the validity property: given a word u such that $v = u(i,p).dec$ is a non-initial and a non-$\bot$ value, adding further states between the initial state and $u(i)$ does not change the fact that v is neither initial nor $\bot$.
- *State invariant* properties, defined by a set S of "good" states, such that for a trace u, $u \in Inv(S)$ only if $\forall i.\ u(i) \in S$. Equivalently, these properties rule out traces which reach the "bad" states in the complement of S. Intuitively, if a bad state is reached, we cannot fix it by adding more states before or after the bad state.

We establish the following lemma, which is essential to the ensuing preservation theorem.

▶ **Lemma 2** (Crash-recovery traces have crash-stop superwords). *Let $\mathcal{A}$ be a crash-stop algorithm with an unreliable failure detector. Let ρ^R be a finite run of the crash-recovery wrapper $\mathcal{A}^R$. Then, there exists a finite run ρ of $\mathcal{A}$ such that $tr_s(\rho^R)$ is a subword of $tr_s(\rho)$, $tr_s(\rho^R)(1) = tr_s(\rho)(1)$, $last(tr_s(\rho)) = last(tr_s(\rho^R))$, no processes fail in ρ, and in-flight messages of $last(\rho)$ match the messages in the buffers of $last(\rho^R)$.*

▶ **Theorem 3** (Preservation of detour-irreparable safety properties). *Let $\mathcal{A}$ be a crash-stop algorithm with an unreliable failure detector, and let P be a safety state property that is not repairable through detours. If $\mathcal{A}$ satisfies P, then so does $\mathcal{A}^R$.*

Proof. We prove the theorem's statement by proving its contrapositive. Assume $\mathcal{A}^R$ violates P. By the definition of safety properties [4], there exists a finite run ρ^R of $\mathcal{A}^R$ such that no continuation of $tr_s(\rho^R)$ is in P. By Lemma 2, there exists a run ρ of $\mathcal{A}$ such that $tr_s(\rho^R)$ is a subword of $tr_s(\rho)$. By Lemma 1, ρ can be extended to an infinite run $\rho \cdot w$ of $\mathcal{A}$. By the choice of ρ^R, we have that $tr_s(\rho^R) \cdot tr_s(w) \notin P$. As $tr_s(\rho^R) \cdot tr_s(w)$ is a subword of $tr_s(\rho \cdot w)$, and since P is not detour repairable, then also $tr_s(\rho \cdot w) \notin P$. Thus, $\mathcal{A}$ also violates P. ◀

Proving that the safety properties of consensus are detour-irrepairable, means that these properties are preserved by our wrapper. Since state invariants are also detour-irreparable, they too are preserved by our wrapper. This makes our wrapper potentially useful for reusing other kinds of crash-stop algorithms in a crash-recovery setting, not just the consensus ones.

▶ **Corollary 4.** *If $\mathcal{A}$ satisfies the safety properties of consensus, then so does $\mathcal{A}^R$.*

▶ **Corollary 5.** *If $\mathcal{A}$ satisfies a state invariant, then so does $\mathcal{A}^R$.*

6 Probabilistic Termination

Termination of consensus algorithms depends on stable periods, during which communication is reliable and no crashes or recoveries occur. In this section, we first state a general result about so-called *selective stable periods* for our probabilistic crash-recovery model. We then define a generic class of crash-stop consensus algorithms, which we call *bounded algorithms*. We prove that termination for these algorithms is guaranteed in our probabilistic model when run under the wrapper. Namely, we prove that, with probability 1, all processes eventually decide. We also show that the class of bounded algorithms covers a wide spectrum of existing algorithms including the celebrated Chandra-Toueg [8] and the instantiation of the indulgent framework of [20] that uses failure detectors.

6.1 Selective Stable Periods

Similar to [11], our proofs will rely on forming strongly-connected communication components between particular sets of processes. However, we will require their existence only for bounded periods of time, which we call selective stable periods.

▶ **Definition 6** (Selective stable period). Fix a crash-recovery algorithm $\mathcal{A}^R$. A *selective-stable period* of $\mathcal{A}^R$ of length Δ for a crash-recovery configuration c and a set of processes C, written $\mathit{stable}(\mathcal{A}^R, \Delta, c, C)$, is the set of all sequences $c = c_0, c_1, \dots c_{\Delta+1}$ of crash-recovery configurations such that $\forall i.\ 1 \le i \le \Delta + 1$ we have $c_i.F = \Pi \setminus C$ and there exist rmsgs_i such that $(c_i, (\mathit{rmsgs}_i, \emptyset), c_{i+1})$ is a step of $\mathcal{A}^R$ and $\mathit{rmsgs}_i(p)(q)$ is defined $\forall p, q \in C$.

Such selective stable periods must occur in runs of a crash-recovery algorithm $\mathcal{A}^R$.

▶ **Lemma 7** (Selective stable periods are mandatory). *Fix a crash-recovery algorithm $\mathcal{A}^R$, a positive integer Δ and a selection function $\mathit{sel} : \mathcal{C}^R \to 2^\Pi$, mapping crash-recovery configurations to process sets. Then, the set of crash-recovery runs*

$$\{c_0, l_0, c_1, \dots \mid \forall i \ge 0.\ c_i, l_i, \dots, c_{i+\Delta+1} \notin \mathit{stable}(\mathcal{A}^R, \Delta, c_i, \mathit{sel}(c_i))\}, \textit{has a probability of } 0.$$

The next section shows how our wrapper exploits such periods to construct crash-recovery algorithms from existing crash-stop ones in a blackbox manner. For future work it might also be interesting to devise consensus algorithm directly on top of this property.

6.2 Bounded Algorithms

We next define the class of *bounded* crash-stop algorithms for which our wrapper guarantees termination in the crash-recovery setting. This class comprises algorithms which operate in rounds, with an upper bound on the number of messages exchanged per round as well the number of rounds correct processes can be apart. More formally, they are defined as follows.

▶ **Definition 8** (Bounded algorithms). A crash-stop consensus algorithm (using reliable links and a failure detector [8]) is said to be *bounded* if it satisfies all properties below:

(B1) **Communication-closed rounds:** processes operate in rounds. The rounds must be communication-closed [14]: only the messages from the current round are considered.
(B2) **Externally triggered state changes:** After the first step of every round, the processes change state only upon receipt of round messages, or on a change of the failure detector output.
(B3) **Bounded round messages:** There exists a bound B_s such that, in any round, a process sends at most B_s messages to any other process.
(B4) **Bounded round gap:** Let $N_c = N - N_f$, that is, the number of correct processes according to the algorithm's resilience criterion. Then, there exists a bound B_Δ, such that the fastest N_c processes are at most B_Δ rounds apart.
(B5) **Bounded termination:** There exists a bound B_{adv} such that for any reachable configuration c where any N_c fastest processes in c are correct, the other processes are faulty, and the failure detector output at these processes is perfect after c, then all of these N_c processes decide before any of them reaches the round $r_{\max}(c) + B_{adv}$.

Checking 1-3 for a given algorithm is typically trivial. We can check 4 by examining under which condition(s) a process increments its round number, and 5 by observing the algorithm's termination under perfect failure detection given a quorum of correct processes. Section 6.3 shows an example of these checks. If an algorithm satisfies the Definition 8, we next prove that it terminates in all sufficiently long selective stable periods.

▶ **Theorem 9** (Bounded selective stable period termination). *Let $\mathcal{A}$ be a bounded algorithm and $\mathcal{A}^R$ its wrapped crash-recovery version. Let c be a reachable crash-recovery configuration of the $\mathcal{A}^R$, and let C be some set of N_c fastest processes in c. Then, there exists a bound B, such that, for any selective stable period of length B for c and C, all processes in C decide in $\mathcal{A}^R$. Moreover, the bound B is independent of the configuration c.*

Proof. We partition the processes from C into the set A (initially C) and NA (initially $\emptyset$). The processes in A will advance their rounds further, and the processes in NA will not advance, but will have already decided. Let p be some slowest process from A in c. We first claim that p advances or decides in any selective stable period for c and C of length at most B_{slow}, defined as $B_{slow} = B_s \cdot B_\Delta + 1$. Denote the period configurations by $c = c_0, c_1, \ldots$. Then, using Lemma 2, we obtain a crash-stop configuration c_1^s from c_1 that satisfies the conditions of bounded termination, with the processes from C being correct, and the others faulty. We consider two cases.

First, if p advances in the crash-stop model after receiving all the in-flight round messages in c_1^s from other processes in A, then it also advances in the crash-recovery model after receiving these messages. Moreover, our wrapper delivers all such messages within $B_s \cdot B_\Delta$ steps, as

(1) it uses LIFO buffers;
(2) bounds B_Δ and B_s apply to c_1^s by 3 and 4; and
(3) by Lemma 2, the same bounds also apply to c_1 (as 4 is an invariant).

Second, if p does not advance in the crash-stop model, then, since the failure detector output remains stable, and since no further round messages will be delivered to p in the crash-stop model, the requirement 2 ensures that p will not advance further in the crash-stop setting. Moreover, requirements 5 and 2 ensure that p must decide after receiving all of its round messages; we move p to the set NA.

We have thus established that the slowest process from A can move to either NA or advance its round after B_{slow} steps. Next, we claim that we can repeat this procedure by picking the slowest member of A again. This is because the procedure ensures that the processes in NA always have round numbers lower than the processes in A. Thus, due to 1 and 2, the processes in A cannot rely on those from NA for changing their state.

Lastly, we note that this procedure needs to be repeated at most $B_{iter} = N \cdot (B_\Delta + B_{adv})$ times before all processes move to the round $r_{max}(c) + B_{adv}$, by which point 5 guarantees that all processes terminate. Thus $B_{slow} \cdot B_{iter}$ gives us the required bound B. ◀

The main result of this paper shows that the wrapper guarantees all consensus properties for wrapped bounded algorithms, including termination.

▶ **Theorem 10** (CR consensus preservation)**.** *If a bounded algorithm $\mathcal{A}$ solves consensus in the crash-stop setting, then $\mathcal{A}^R$ also solves consensus in the probabilistic crash-recovery setting.*

Proof. By Corollary 4, we conclude that $\mathcal{A}^R$ solves the safety properties of consensus in the crash-recovery setting. For (probabilistic) termination, the result follows from Theorem 9 and Lemma 7, using as the selection function for Lemma 7

(1) any function that selects some N_c fastest processes in a configuration if no process has decided yet and

(2) all processes, if some process has decided.

The latter allows us to propagate the decision to all processes, due to short-circuiting in $\mathcal{A}^R$. ◀

6.3 Examples of Bounded Algorithms

We next give two prominent examples of bounded algorithms: the Chandra-Toueg (CT) algorithm [8] and the instantiation of the indulgent framework of [20] that uses failure detectors. For these algorithms, rounds are composite, and consist of the combination of what the authors refer to as rounds and phases. Checking that the algorithms then satisfy conditions 1–3 is straightforward. 4 holds for the CT algorithm with $B_\Delta = 4 \cdot N$: take the fastest process p in a crash-stop configuration; if its CT round number r_p is N or less, the claim is immediate. Otherwise, p must have previously moved out of the phase 2 of the last round r'_p in which it was the coordinator, which implies that at least N_c processes have also already executed r_p. Since CT uses the rotating coordinator paradigm, $r_p - r'_p \leq N$; as each round consists of 4 phases, $B_\Delta = 4 \cdot N$. For the algorithm from [20], processes only advance to the next round (which consists of two phases) when they receive messages from N_c other processes. Thus, $B_\Delta = 2$. Finally, proving the requirement 5 is similar to, but simpler than the original termination proofs for the algorithms, since it only requires termination under conditions which includes perfect failure detector output. For space reasons, we do not provide the full proofs here, but we note that $B_{adv} = \mathit{phases} \cdot \lfloor N/2 \rfloor$ for both algorithms, where *phases* is the number of phases per algorithm round. Intuitively, within this many rounds the execution hits a round where a correct processor is the coordinator; since we assume perfect failure detection for this period, no process will suspect this coordinator, and thus no process will move out of this round without deciding.

▶ **Corollary 11.** *The wrapped versions of Chandra-Toueg's algorithm [8] and the instantiation of the indulgent framework of [20] using failure detectors solve consensus in the probabilistic crash-recovery model.*

7 Concluding Remarks

This paper introduced new system models that closely capture the messy reality of distributed systems. Unlike the usual distributed computing models, we account for failure and recovery patterns of processes and links in a probabilistic and temporary, rather than a deterministic and perpetual manner. Our models allow an unbounded number of processes and communication links to probabilistically fail and recover, potentially for an infinite number of times. We showed how and under which conditions we can reuse existing crash-stop distributed algorithms in our crash-recovery systems. We presented a wrapper that allows crash-stop algorithms to be deployed unchanged in our crash-recovery models. The wrapper preserves the correctness of a wide class of consensus algorithms.

Our work opens several new directions for future investigations. First, we currently model failures of processes as well as communication links individually and independently, with a non-zero probability of failing/recovering at any point in time. In the full version, we sketch how our results can be extended to systems where some processes may never even recover from failure. It is interesting to investigate what results can be established with more complicated probability distributions, e.g., if the model is weakened to allow processes and links to fail/recover on average with some non-zero probability [15]. Second, our wrapper fully persists the processes state. Studying how to minimize the amount of persisted state while still allowing our results (or similar ones) to hold is another promising direction. Finally, we focus on algorithms that depend on the reliability of message delivery. Some algorithms, notably Paxos [23], do not. Finding a modular link abstraction for the crash-stop setting that identifies these algorithms is another interesting topic. For those algorithms, we speculate that preserving termination in the crash-recovery model is simpler.

References

1 Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Failure detection and consensus in the crash-recovery model. *Distributed computing*, 13(2):99–125, 2000.

2 Dan Alistarh, James Aspnes, Valerie King, and Jared Saia. Communication-Efficient Randomized Consensus. In *Distributed Computing*, pages 61–75, 2014.

3 Dan Alistarh, Seth Gilbert, Rachid Guerraoui, and Corentin Travers. How to solve consensus in the smallest window of synchrony. In *DISC*. Springer, 2008.

4 Bowen Alpern and Fred Schneider. Defining Liveness. *Information Processing Letters*, 21:181–185, June 1985.

5 James Aspnes, Hagit Attiya, and Keren Censor. Combining Shared-coin Algorithms. *J. Parallel Distrib. Comput.*, 70(3):317–322, 2010.

6 Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

7 Gabriel Bracha and Sam Toueg. Asynchronous Consensus and Broadcast Protocols. *J. ACM*, 32(4), 1985.

8 Tushar Deepak Chandra and Sam Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *J. ACM*, 43(2):225–267, 1996.

9 Bernadette Charron-Bost, Martin Hutle, and Josef Widder. In search of lost time. *Information Processing Letters*, 110(21), 2010.

10 Flavin Cristian. Understanding Fault-tolerant Distributed Systems. *Commun. ACM*, 34(2):56–78, 1991.

11 Danny Dolev, Roy Friedman, Idit Keidar, and Dahlia Malkhi. Failure Detectors in Omission Failure Environments. In *PODC*, pages 286–, 1997.

12 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the Presence of Partial Synchrony. *J. ACM*, 35(2):288–323, 1988.

13 Dacfey Dzung, Rachid Guerraoui, David Kozhaya, and Yvonne Anne Pignolet. Never Say Never - Probabilistic and Temporal Failure Detectors. In *IEEE International Parallel and Distributed Processing Symposium, IPDPS*, pages 679–688, 2016.

14 Tzilla Elrad and Nissim Francez. Decomposition of distributed programs into communication-closed layers. *Science of Computer Programming*, 2:155–173, 1982.

15 Christof Fetzer, Ulrich Schmid, and Martin Susskraut. On the Possibility of Consensus in Asynchronous Systems with Finite Average Response Times. In *25th IEEE International Conference on Distributed Computing Systems*, pages 271–280, 2005.

16 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.

17 Pierre Fraigniaud, Mika Göös, Amos Korman, Merav Parter, and David Peleg. Randomized distributed decision. *LNCS*, 27, 2014.

18 Felix C. Freiling, Christian Lambertz, and Mila Majster-Cederbaum. Modular Consensus Algorithms for the Crash-Recovery Model. In *2009 International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 287–292, December 2009.

19 Eli Gafni. Round-by-round fault detectors: Unifying synchrony and asynchrony. In *PODC*, pages 143–152, 1998.

20 Rachid Guerraoui and Michel Raynal. A Generic Framework for Indulgent Consensus. In *ICDCS*, pages 88–, 2003.

21 Michel Hurfin, Achour Mostéfaoui, and Michel Raynal. Consensus in Asynchronous Systems Where Processes Can Crash and Recover. In *Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems*, pages 280–, 1998.

22 Michel Hurfin, Achour Mostéfaoui, and Michel Raynal. A versatile family of consensus protocols based on Chandra-Toueg's unreliable failure detectors. *IEEE Transactions on Computers*, 51(4):395–408, 2002.

23 Leslie Lamport. The Part-time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

24 Mikel Larrea, Cristian Martín, and Iratxe Soraluze. Communication-efficient leader election in crash–recovery systems. *Journal of Systems and Software*, 84(12):2186–2195, 2011.

25 Neeraj Mittal, Kuppahalli L. Phaneesh, and Felix C. Freiling. Safe Termination Detection in an Asynchronous Distributed System when Processes May Crash and Recover. *Theor. Comput. Sci.*, 410(6-7):614–628, February 2009.

26 H. Moniz, N.F. Neves, and M. Correia. Turquois: Byzantine consensus in wireless ad hoc networks. In *DSN*, 2010.

27 Henrique Moniz, NunoFerreira Neves, Miguel Correia, and Paulo Veríssimo. Randomization Can Be a Healer: Consensus with Dynamic Omission Failures. In *DISC*, volume 5805 of *LNCS*, 2009.

28 Rui Oliveira, Rachid Guerraoui, and André Schiper. Consensus in the Crash-Recover Model, 1997.

29 Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.

30 André Schiper. Early Consensus in an Asynchronous System with a Weak Failure Detector. *Distrib. Comput.*, 10(3):149–157, 1997.

31 Ulrich Schmid, Bettina Weiss, and Idit Keidar. Impossibility results and lower bounds for consensus under link failures. *SIAM Journal on Computing*, 38(5):1912–1951, 2009.

Causal Broadcast: How to Forget?

Brice Nédelec
LS2N, University of Nantes, 2 rue de la Houssinière, BP 92208, 44322 Nantes Cedex 3, France
brice.nedelec@ls2n.fr

Pascal Molli
LS2N, University of Nantes, 2 rue de la Houssinière, BP 92208, 44322 Nantes Cedex 3, France
pascal.molli@ls2n.fr

Achour Mostéfaoui
LS2N, University of Nantes, 2 rue de la Houssinière, BP 92208, 44322 Nantes Cedex 3, France
achour.mostefaoui@ls2n.fr

Abstract

Causal broadcast constitutes a fundamental communication primitive of many distributed protocols and applications. However, state-of-the-art implementations fail to forget obsolete control information about already delivered messages. They do not scale in large and dynamic systems. In this paper, we propose a novel implementation of causal broadcast. We prove that all and only obsolete control information is safely removed, at cost of a few lightweight control messages. The local space complexity of this protocol does not monotonically increase and depends at each moment on the number of messages still in transit and the degree of the communication graph. Moreover, messages only carry a scalar clock. Our implementation constitutes a sustainable communication primitive for causal broadcast in large and dynamic systems.

2012 ACM Subject Classification Computer systems organization → Peer-to-peer architectures

Keywords and phrases Causal broadcast, complexity trade-off, large and dynamic systems

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.20

Funding This work was funded by the French ANR projects O'Browser (ANR-16-CE25-0005-01), and Descartes (ANR-16-CE40-0023).

1 Introduction

Causal broadcast constitutes the core communication primitive of many distributed systems [9]. Applications such as distributed social networks [3], distributed collaborative software [10, 19], or distributed data stores [2, 4, 6, 15, 24] use causal broadcast to ensure consistency criteria. Causal broadcast ensures reliable receipt of broadcast messages, exactly-once delivery, and causal delivery following Lamport's happen before relationship [14]. When Alice comments on Bob's picture, nobody sees Alice's comment without Bob's picture, and nobody sees multiple occurrences of Alice's comment or Bob's picture.

Vector clock-based approaches [16, 18, 20] need to keep all their control information forever. They cannot forget any control information. The consumed memory monotonically increases with the number of processes that ever broadcast a message $O(N)$. They become unpractical in large and dynamic system comprising from hundreds to millions of processes joining, leaving, self-reconfiguring, or crashing at any time.

In this paper, we propose a novel implementation of causal broadcast that forgets all and only obsolete control information. A process p that has i incoming links receives each message i times. A message m is active for p between its first and last reception by p. Process p keeps control information about all its active messages. As soon as a message becomes

© Brice Nédelec, Pascal Molli, and Achour Mostéfaoui;
licensed under Creative Commons License CC-BY

22nd International Conference on Principles of Distributed Systems (OPODIS 2018).
Editors: Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira; Article No. 20; pp. 20:1–20:16

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

inactive, the process can forget all control information related to it. Consequently, processes do no store any permanent control information about messages. When no message is active, no control information is stored in the system.
Our contribution is threefold:

- We define the notion of *link memory* as a mean for each process to forbid multiple delivery. Link memory allows each process to identify processes from which it will receive a copy of an already delivered message. This allows each process to safely remove obsolete control information about broadcast messages that will never be received again. We prove that using causal delivery, each process can build such knowledge even in dynamic systems where processes may join, leave, self-reconfigure, or crash at any time.
- We propose an implementation of causal broadcast that uses the notion of link memory, where each process manages a local data the size of which is $O(i \cdot A)$ where i is the number of incoming links and A is the number of active messages. Moreover, the only control information piggybacked on messages is a scalar Lamport clock.
- We evaluate our implementation using large scale simulations. The experiments highlight the space consumed and the traffic generated by our protocol in dynamic systems with varying latency. The results confirm that the proposed approach scales with system settings and use.

The rest of this paper is organized as follows. Section 2 describes the model, highlights the issue, introduces the principle solving the issue, provides an implementation solving the issue along with its complexity analysis. Section 3 shows the experiments. Section 4 reviews related work. We conclude and discuss about perspectives in Section 5.

2 Proposal

In this section, we present a causal broadcast protocol providing a novel trade-off between speed, memory, and traffic. Among others, it safely removes obsolete control information about broadcast messages. Its memory consumption increases and decreases over receipts. The key ideas are:

(1) Every process broadcasts or forwards a message once, hence every link carries a message once. A process expects to receive as many copies of a message as its number of links. Once a process received all expected copies, it can forget about this broadcast message, i.e., it can safely remove the control information associated to this broadcast message.

(2) Adding links between pair of processes adds uncertainty. The receiver cannot state any longer when it should expect a message from a link.

(3) By exploiting causal order, processes remove this uncertainty. Causal order allows processes to remove batches of obsolete information while reasoning about temporarily buffered broadcast messages.

2.1 Model

A distributed system comprises a set of processes that can communicate with each other using messages. Processes may not have the knowledge of all processes in the system. Instead, processes build and maintain overlay networks: each process updates a local partial view of logical communication links, i.e., a set of processes to communicate with. The partial view is usually much smaller than the actual system size. We use the terms of overlay networks, and distributed systems interchangeably.

▶ **Definition 1** (Overlay network)**.** An overlay network $G = (P, E)$ comprises a set of processes P and a set of directed links $E \subseteq P \times P$. An overlay network is static if both sets P and E are immutable. Otherwise, the overlay network is dynamic. An overlay network is strongly connected if there exists a path – i.e. a link or an sequence of links – from any process to any other process. We only consider strongly connected overlay networks.

▶ **Definition 2** (Process)**.** A process runs a set of instructions sequentially. Processes communicate with each other using asynchronous message passing. A process A can send a message to a process B $s_{AB}(m)$, or to any process $s_A(m)$; receive a message from a process B $r_{AB}(m)$, or from any process $r_A(m)$. A process sends messages using the set of links departing from it, called out-view Q_o. Processes reachable via these links are called neighbors. A process receives messages from the set of links arriving to it, called in-view Q_i. Processes are faulty if they crash, otherwise they are correct. We do not consider Byzantine processes.

Causal broadcast ensures properties similar to those of reliable broadcast. Each process may receive each broadcast message multiple times but delivers it once. In this paper, we tackle the issue of implementing these properties.

▶ **Definition 3** (Uniform reliable broadcast)**.** When a process A broadcasts a message to all processes of its system $b_A(m)$, each correct process B eventually receives it and delivers it $d_B(m)$. Uniform reliable broadcast guarantees 3 properties:

(1) Validity: If a correct process broadcasts a message, then it eventually delivers it.

(2) Uniform Agreement: If a process – correct or not – delivers a message, then all correct processes eventually deliver it.

(3) Uniform Integrity: A process delivers a message at most once, and only if it was previously broadcast.

In static systems, implementing these properties only requires a local structure the size of which grows and shrinks over receipts [22]. Every process knows the number of copies of each delivered message it should expect. When this number drops down to 0, the process safely removes the control information associated to the delivered messages. This forbids multiple delivery for the process will never receive – hence deliver – this message again.

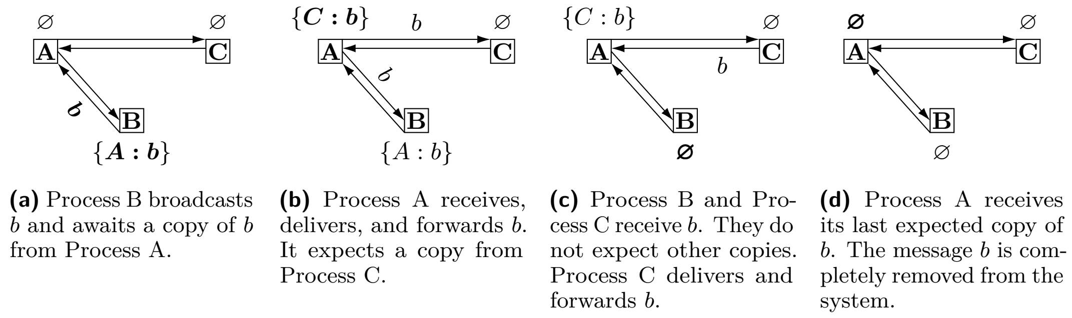
However, in dynamic systems where processes join, leave, or self-reconfigure their out-view at any time, the removal of a link may impair the consistency of the number of expected messages. Either processes cannot safely garbage collect obsolete control information, or processes suffer from multiple delivery.

In this paper, we solve this issue by exploiting causal broadcast's ability to ensure a specific order on message delivery. To characterize the order among events such as send, or receive, we define time in a logical sense using Lamport's definition.

▶ **Definition 4** (Happens-before [14])**.** The happens-before relationship defines a strict partial order of events. The happens-before relationship $\rightarrow$ is transitive ($e_1 \rightarrow e_2 \wedge e_2 \rightarrow e_3 \implies e_1 \rightarrow e_3$), irreflexive ($e_1 \not\rightarrow e_1$), and antisymmetric ($e_1 \rightarrow e_2 \implies e_2 \not\rightarrow e_1$). The sending of a message always precedes its receipt $s_{AB}(m) \rightarrow r_{BA}(m)$. Two messages are concurrent if none happens before the other ($r_A(m_1) \not\rightarrow s_A(m_2) \wedge r_A(m_2) \not\rightarrow s_A(m_1)$).

▶ **Definition 5** (Causal order)**.** The delivery order of messages follows the happen before relationships of the corresponding broadcasts. $d_A(m) \rightarrow b_A(m') \implies d_B(m) \rightarrow d_B(m')$

▶ **Definition 6** (Causal broadcast)**.** Causal broadcast is a uniform reliable broadcast ensuring causal order.



(a) Process B broadcasts b and awaits a copy of b from Process A.

(b) Process A receives, delivers, and forwards b. It expects a copy from Process C.

(c) Process B and Process C receive b. They do not expect other copies. Process C delivers and forwards b.

(d) Process A receives its last expected copy of b. The message b is completely removed from the system.

Figure 1 Link memory allows to safely remove obsolete control information in static systems.

2.2 Link memory

We define link memory as a mean for processes to forbid multiple delivery while safely removing obsolete control information. Processes attach control information about expected messages to each link of their respective in-view.

▶ **Definition 7** (Link memory). Assuming a link (A, B), Process B remembers among its delivered messages those that it will receive from this link; and forgets among its delivered messages those that it will never receive from this link. $remember_{BA}(m) \equiv d_B(m) \wedge \neg r_{BA}(m)$

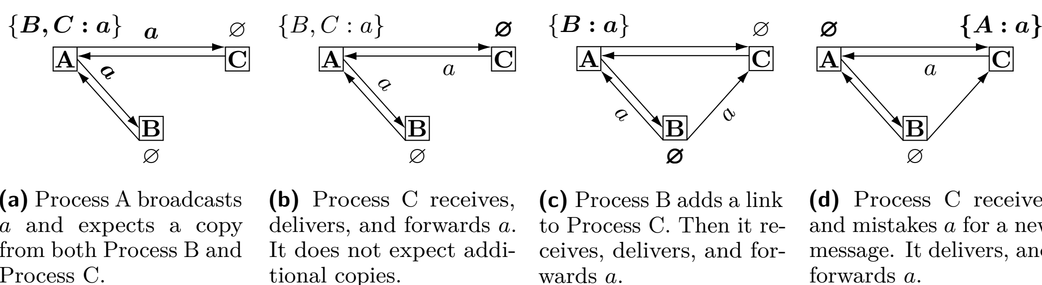
▶ **Theorem 8** (Link memory forbids multiple delivery). *A process that delivers only messages it does not remember using link memory delivers each broadcast message exactly once.*

Proof. We must show that, for any message m, its receipt cannot lead to its delivery if a copy of m has already been delivered before: $\nexists m,\, d_B(m) \rightarrow r_{BA}(m) \wedge r_{BA}(m) \rightarrow d_B(m)$.
The delivery $d_B(m)$ implies a prior receipt $r_B(m) \rightarrow d_B(m)$. Assuming that each link carries each messages once, and this receipt comes from link (A, B), then Process B cannot receive, hence deliver, m from (A, B) again: $\nexists m,\, r_{BA}(m) \rightarrow d_B(m) \wedge d_B(m) \rightarrow r_{BA}(m) \wedge r_{BA}(m) \rightarrow d_{BA}(m)$.
If this receipt comes from any other link (C, B), $C \neq A$, Process B remembers m on other links, and among others, on link (A, B): $\forall m,\, r_{BC}(m) \rightarrow remember_{BA}(m)$. Assuming that each link carries each message once, Process B eventually receives a copy of m from link (A, B): $remember_{BA}(m) \rightarrow r_{BA}(m)$. Since remembering a message forbids its delivery, this cannot lead to another delivery of m: $\nexists m,\, r_{BA}(m) \rightarrow d_B(m)$.
Finally, since every process delivers, hence forwards each message once, each link carries each messages once. ◀

Algorithm 1 shows a set of instructions that implements causal broadcast for static systems. It uses reliable FIFO links to ensure causal order [7], and implements link memory to forbid multiple delivery. Every process maintains a local structure the size of which increases and decreases over receipts. The first receipt of a broadcast message from a link tags the other links (see Line 15). The receipt on other links of this broadcast message removes the corresponding tag (see Line 16). Figure 1 depicts its functioning in a system comprising 3 processes. In Figure 1a, Process B broadcasts b. It awaits a copy of b from the only link in its in-view. In Figure 1b, Process A receives b. It delivers it, for no link in its in-view is tagged with b, meaning this is a first receipt. It tags the other link in its in-view with b and forwards b to its out-view. In Figure 1c, Process B receives the awaited copy of b from Process A. It removes the corresponding entry. The broadcast protocol at

Algorithm 1: Causal broadcast for static systems.

```
1  Q_o                                                    // Out-view
2  Q_i                                                    // In-view
3  E ← ∅                          // Map of expected messages Q_i : M*
4  DISSEMINATION:
5     function C-broadcast(m)
6        receive(m, _)
7     upon receive(m, l)
8        if ¬received(m, l) then
9           foreach q ∈ Q_o do sendTo(q, m)
10          C-deliver(m)
11    function received(m, l)
12       rcvd ← ∃q ∈ E with m ∈ E[q]
13       if ¬rcvd then
14          foreach q ∈ Q_i do
15             E[q] ← E[q] ∪ m
16       E[l] ← E[l] \ m
17       return rcvd
```



(a) Process A broadcasts a and expects a copy from both Process B and Process C.

(b) Process C receives, delivers, and forwards a. It does not expect additional copies.

(c) Process B adds a link to Process C. Then it receives, delivers, and forwards a.

(d) Process C receives and mistakes a for a new message. It delivers, and forwards a.

Figure 2 Causal broadcast (Algorithm 1) fails to forbid multiple delivery in dynamic systems.

Process B does not consume space anymore. Process C receives b. It detects a first receipt so it delivers and forwards b. It does not tag any link, for the only link from its in-view is the link from which it just received b. In Figure 1d, the last process to await a copy of b finally receives it. None of processes remembers about b. No copy of b travels in the system. This implementation forbids multiple delivery in static systems while safely removing obsolete control information.

However, implementing link memory becomes more challenging in dynamic systems where processes can start sending messages to any other process at any time. Any process can receive an already forgotten message from any other process. Figure 2 illustrates the issue. In Figure 2a, Process A broadcasts a. It expects a copy from both Process B and Process C. In Figure 2b, Process C immediately receives, delivers, and forwards a. It does not tag any link and expects to never receive this message again. However, network condition delays the receipt of b from Process B. In Figure 2c, Process B adds a communication link towards Process C. Then it receives, delivers, and forwards b. Since Process C now belongs to its out-view, the forwarding includes Process C. In Figure 2d, Process C receives a again. However, it did not keep control information about this message. It mistakes it for a first receipt. It delivers and forwards a. Not only Process C suffers multiple delivery but this has cascading effects over the whole system.

The rest of this section describes how causal broadcast can exploit causal order to initialize link memory, an implementation of such broadcast, and its complexity analysis.

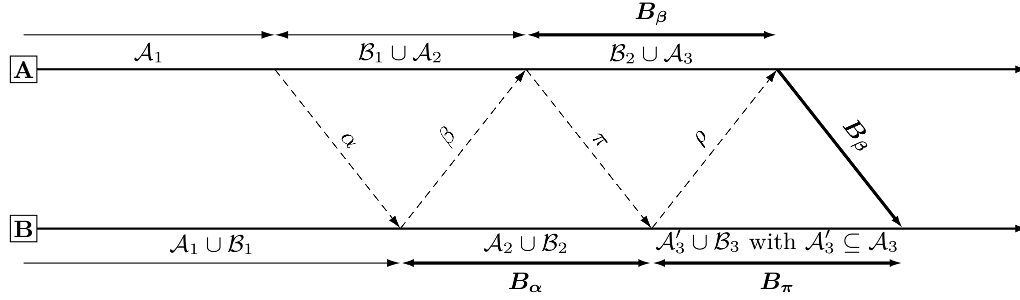


Figure 3 Initializing the link memory from Process A to Process B. Control messages α, β, π, and ρ are delivered after all preceding messages while B_β is not. At receipt of B_β, Process B classifies the messages: $B_\beta \cap (B_\alpha \cup B_\pi) = \mathcal{B}_2 \cup \mathcal{A}'_3$ are messages to ignore; $B_\beta \setminus B_\alpha \setminus B_\pi = \mathcal{A}_3 \setminus \mathcal{A}'_3$ are messages to deliver; $B_\pi \setminus B_\beta = \mathcal{B}_3$ are messages to expect from Process A.

2.3 Link memory for dynamic systems

This section demonstrates that causal broadcast can use causal order to initialize link memory, thereby enabling the use of link memory in dynamic systems.

Algorithm 1 already implements the maintenance of link memory over receipts. Every process safely removes obsolete control information over receipts. However, Figure 2 highlights that new links lack of consistent initialization. The challenge consists in initializing such memory without history of past messages. Causal broadcast starts to build the knowledge on-demand, i.e., when a process wants to add a link to another process. The protocol disables the new link until initialized. This initialization requires round-trips of control messages and message buffering. Causal broadcast takes advantage of causal order to provide guarantees on messages included in buffers.

Figure 3 depicts the principle of the approach. When a Process A adds a link to Process B, Process A notifies Process B using a control message α. This control message α, as all control messages that will follow (β, π, ρ), must be delivered after all its preceding messages. Hence, at receipt, Process B implicitly removes obsolete information: messages delivered by Process A before the sending of the notification $\mathcal{A}_1$. At receipt of α, Process B can start gathering control information about its delivered messages in a buffer B_α. Among other, Process B wants to identify messages concurrent to the correct establishment of the new link. Process B acknowledges Process A's notification using a control message β. At receipt of β, Process A removes obsolete information: messages delivered by Process B before the sending of the acknowledgment $\mathcal{A}_1 \cup \mathcal{B}_1$. This solves the issue identified in Figure 2, for a would belong to $\mathcal{A}_1$ or $\mathcal{B}_1$. However, this is not sufficient to initialize link memory. Process A sends a control message π to Process B, and starts to gather control information about its delivered messages in a buffer B_β. Upon receipt of π, Process B closes its first buffer B_α.

▶ **Lemma 9** (Messages in buffer B_α). *The buffer B_α contains messages delivered by Process B after the sending of β and before the receipt of π.*
This includes all messages delivered by Process A before the sending of π that were not delivered by Process B before the sending of β: $\mathcal{A}_2$. Above all, this also includes all messages delivered by Process B that were not delivered by Process A at the sending of π: $\mathcal{B}_2$.

Proof. Since control messages are delivered after preceding messages, all broadcast messages delivered by Process A before the sending of α precede the buffering: $\forall m,\ d_A(m) \rightarrow s_A(\alpha_{AB}) \implies m \notin B_\alpha$. Since messages are delivered once, $\forall m,\ d_A(m) \rightarrow s_A(\pi_{AB}) \wedge d_B(m) \rightarrow s_B(\beta_{AB}) \implies m \notin B_\alpha$. This removes $\mathcal{A}_1$ and $\mathcal{B}_1$.

The buffer B_α contains the rest of messages delivered by Process B before the receipt of π. This includes messages delivered by Process A between the sending of α and π but not delivered by Process B before the sending of β ($\mathcal{A}_2$); and messages delivered by Process B but not delivered by Process A before the sending of π ($\mathcal{B}_2$). ◀

Upon receipt of π, Process B continues to gather control information about its delivered messages in another buffer B_π. Some messages in this buffer will be expected from Process A, but Process B cannot determine which ones just yet. It sends the last acknowledgment ρ to Process A. Upon receipt of this acknowledgment, Process A closes its buffer and sends it using the new link. Afterwards, Process A uses the new link for causal broadcast, for it knows that Process B will receive B_β before upcoming broadcast messages on this new link, and the receipt of B_β will allow Process B to initialize this new link memory.
Upon receipt of B_β, Process B stops buffering in B_π.

▶ **Lemma 10** (Messages in buffer B_β). *The buffer B_β contains messages delivered by Process A after the sending of π and before the receipt of ρ.*
This includes all messages delivered by Process B before the sending of ρ that were not delivered by Process A before the sending of π: $\mathcal{B}_2$. This also includes all messages delivered by Process A that were not delivered by Process B at the sending of ρ: $\mathcal{A}_3$.

Proof. The proof is similar to that of Lemma 9. Control messages shift roles. π becomes ρ; β becomes π; α becomes β. ◀

▶ **Lemma 11** (Messages in buffer B_π). *The buffer B_π contains messages delivered by Process B after the sending of π and before the receipt of B_β.*
This may includes messages delivered by Process A before the sending of B_β that were not delivered by Process B before the sending of ρ: $\mathcal{A}'_3$. This also includes all messages delivered by Process B that were not delivered by Process A at the sending of B_β: $\mathcal{B}_3$.

Proof. The proof is similar to that of Lemmas 10 and 11. The difference being that B_β is not necessarily delivered after preceding messages. Hence, the receipt of B_β follows the sending of ρ but Process B cannot state if it received all, part, or none of messages in $\mathcal{A}_3$. Thus, $\mathcal{A}'_3 \subseteq \mathcal{A}_3$. ◀

Using B_α, B_β, and B_π buffers, Process B identifies messages in B_β it must deliver against messages it must ignore, and messages in B_π it must receive from Process A. This allows Process B to initialize link memory.

▶ **Theorem 12** (B_α, B_β, and B_π initialize link memory). *A process consistently initializes link memory at receipt of B_β using B_α and B_π.*

Proof. From Lemma 9, $B_\alpha = \mathcal{A}_2 \cup \mathcal{B}_2$. From Lemma 10, $B_\beta = \mathcal{B}_2 \cup \mathcal{A}_3$. From Lemma 11, $B_\pi = \mathcal{A}'_3 \cup \mathcal{B}_3$.
First, we must show that Process B delivers all and only messages from B_β it did not deliver yet: $m \in \mathcal{A}_3 \setminus \mathcal{A}'_3$.
Since $B_\beta \setminus B_\alpha \setminus B_\pi = (\mathcal{B}_2 \cup \mathcal{A}_3) \setminus (\mathcal{A}_2 \cup \mathcal{B}_2) \setminus (\mathcal{A}'_3 \cup \mathcal{B}_3) = \mathcal{A}_3 \setminus (\mathcal{A}'_3 \cup \mathcal{B}_3)$. Since $\mathcal{B}_3 \cap \mathcal{A}_3 = \varnothing$, we have $B_\beta \setminus B_\alpha \setminus B_\pi = \mathcal{A}_3 \setminus \mathcal{A}'_3$.
Second, we must show that Process B initializes the new link memory with all and only messages from B_π that Process A did not deliver at the sending of B_β: $m \in \mathcal{B}_3$.
$B_\pi \setminus B_\beta = (\mathcal{A}'_3 \cup \mathcal{B}_3) \setminus (\mathcal{B}_2 \cup \mathcal{A}_3)$. Since $\mathcal{B}_3 \cap \mathcal{B}_2 = \varnothing$ and $\mathcal{A}'_3 \subseteq \mathcal{A}_3$, $B_\pi \setminus (B_\beta \setminus B_\alpha) = \mathcal{B}_3$. ◀

Algorithm 2: PRC-broadcast at Process p.

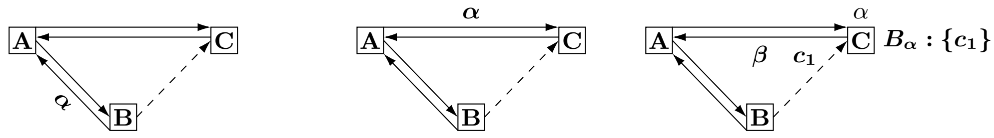
```
1  B ← ∅                                   // @Sender Map of buffers Q_o : M*
2  S ← ∅                         // @Receiver Map of buffers Q_i : M* × M* × bool
3  DISSEMINATION:
4     function PRC-broadcast(m)
5        C-broadcast(m)
6     upon C-deliver(m)
7        buffering(m)
8        PRC-deliver(m)
9     function buffering(m)
10       foreach q ∈ B do B[q] ← B[q] ∪ m
11       foreach ⟨B_α, B_π, received_π⟩ ∈ S do
12          if received_π then B_π ← B_π ∪ m
13          else B_α ← B_α ∪ m
14 LINK MEMORY:
      @Sender
15    upon open_o(to)
16       Q_o ← Q_o \ to
17       send-α(p, to)
20    upon receive-β(from, to)
21       B[to] ← ∅
22       send-π(from, to)
26    upon receive-ρ(from, to)
27       send-B_β(from, to, B[to])
28       B ← B \ to
29       Q_o ← Q_o ∪ to
41    upon close_o(to)
42       B ← B \ to

      @Receiver
18    upon open_i(from)
19       Q_i ← Q_i \ from
23    upon receive-α(from, to)
24       S[from] ← ⟨∅, ∅, false⟩
25       send-β(from, to)
30    upon receive-π(from, to)
31       ⟨B_α, B_π, _⟩ ← S[from]
32       S[from] ← ⟨B_α, B_π, true⟩
33       send-ρ(from, to)
34    upon receive-B_β(from, to, B_β)
35       ⟨B_α, B_π, _⟩ ← S[from]
36       S ← S \ from
37       foreach m ∈ B_β \ B_α \ B_π do          filter messages to ignore →
38          receive(m, from)                      to deliver →
39       E[from] ← B_π \ B_β                      to expect →
40       Q_i ← Q_i ∪ from
43    upon close_i(from)
44       S ← S \ from
45       E ← E \ from
```

2.4 Implementation

PRC-broadcast stands for Preventive Reliable Causal broadcast. It prevents both causal order violations and multiple delivery by using all and only links that are safe [20], and the memory of which is correctly initialized and maintained. PRC-broadcast ensures that control messages are delivered after all their preceding messages by sending them on reliable FIFO links used for causal broadcast. PRC-broadcast uses a local structure the size of which increases and decreases over receipts. Every process safely removes obsolete control information about past broadcast messages.

Algorithm 2 shows the instructions of PRC-broadcast. Figure 4 illustrates its operation in a scenario involving 3 processes. In this example, Process B adds a link to Process C. Process B disables the new link for causal broadcast until it is safe and guaranteed that Process C correctly initialized its memory.
Process B sends a first control message α to Process B using safe links (see Line 17 and Figure 4a).
After being routed to Process C by intermediary processes (see Figure 4b), α reaches Process C (see Figure 4c). Process C starts to register messages it delivers in a buffer B_α. Process C



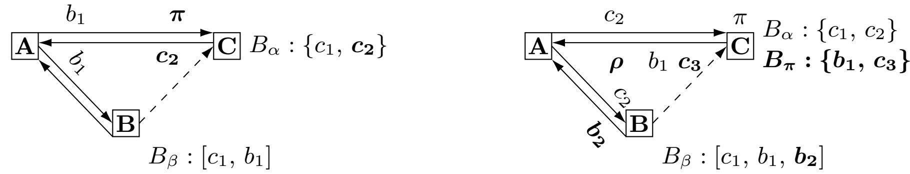
(a) Process B adds a link to Process C. PRC-broadcast ensures its safety. Process B sends a first control message α to Process C using Process A as mediator.

(b) Process A receives α and routes it to Process C.

(c) Process C receives α and answers by sending β to Process B using Process A as mediator. Then, Process C broadcasts c_1 and registers it in B_α.

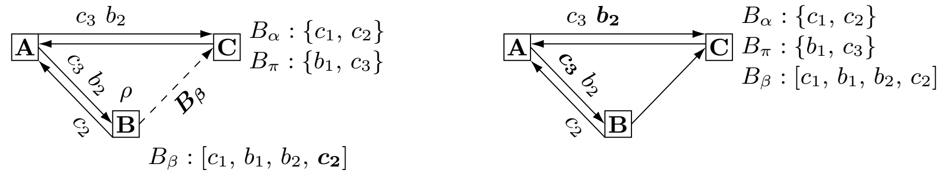
(d) Process A receives β and routes it to Process B. Process A receives c_1 and forwards it to both its neighbors.

(e) Process C receives and discards c_1. Process B receives β and replies π to Process C using Process A as mediator. Process B receives c_1 and forwards it to its neighbor. Process B broadcasts b_1. It registers c_1 and b_1 in B_β.



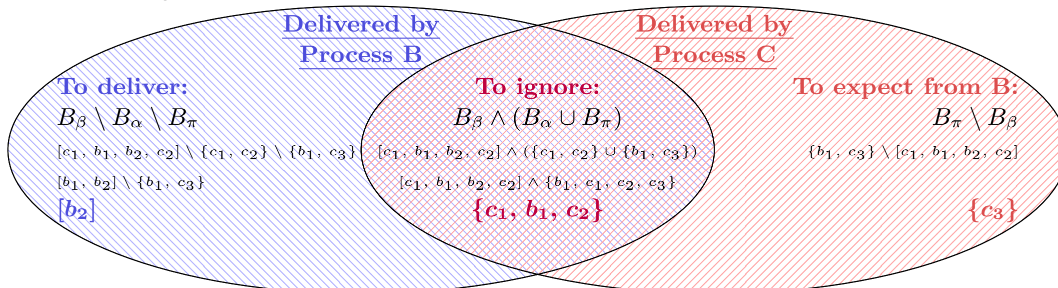
(f) Process A receives c_1 and discards it. Process A receives π and routes it to Process C. Process A receives b_1 and forwards it to its neighbors. Process C broadcasts c_2 and registers it in B_α.

(g) Process A receives c_2 and forwards it to its neighbors. Process B broadcasts b_2 and registers it in B_β. Process C receives π and replies ρ to Process B using Process A as mediator. Then it receives and forwards b_1. Then it broadcasts c_3. It registers b_1 and c_3 in B_π.



(h) Process A receives and discards b_1. Process A receives and routes ρ to Process B. Process A receives and forwards b_2 then c_3. Process B receives, forwards, and registers c_2. Then Process B receives ρ and sends B_β to Process C using the new link.

(i) Once Process A sent B_β, the new link is safe. Process C receives B_β. Process C does not deliver c_1, b_1 and c_2, for it already delivered them. Process C delivers b_2 and expects another copy from Process A, for it constitutes a new message. Process C expects to eventually receive c_3 from Process B.



(j) Process C categorizes each message of B_β and B_π.

Figure 4 Using buffers and control messages, PRC-broadcast provides reliable causal broadcast.

acknowledges the receipt of α by sending a second control message β to Process B using safe links (see Line 25). In Figure 4c, Process C broadcasts c_1 and registers it in B_α.
After being routed to Process B (see Figure 4d), β reaches Process B. Process B starts to register messages it delivers in a buffer B_β. Process B sends a third control message π to Process C using safe links (see Line 22). In Figure 4e, Process B delivers c_1 then broadcasts b_1. It registers them in B_β. In Figure 4f, Process C broadcasts c_2 and registers it in B_α.
After being routed to Process C by intermediary processes (see Figure 4f), π reaches Process C. Process C ends its first buffer B_α. Process C starts to register messages it delivers in B_π. Process C sends a fourth and last control message ρ to Process B using safe links (see Line 33). In Figure 4g Process C delivers b_1, broadcasts c_3, and registers them in B_π. In the meantime, Process B broadcasts b_2 and registers it in B_β.
After being routed to Process B, ρ reaches Process B (see Figure 4h). Process B stops buffering and sends its buffer of messages B_β using the new link $s_{BC}(B_\beta)$ (see Line 27). In Figure 4i, this buffer contains b_1, b_2, and c_2. The new link is safe. Process B starts to use this link normally for causal broadcast using Algorithm 1.
Once Process C receives the buffer, it ends its buffer B_π (see Figure 4i). Using B_α, and B_π, Process C identifies among messages from B_β the array of messages to deliver (see Line 38). In Figure 4j, this array only includes b_2. Using B_α, and B_π, Process C also identifies the set of messages to ignore which is the rest of the buffer. In Figure 4j, this set includes c_1, b_1 and c_2. Finally, Process C identifies among its own delivered messages the messages to expect from Process B (see Line 39). In Figure 4j, this set includes c_3. This set constitutes the memory of the new safe link. Afterwards, messages received by this new link are processed normally.

PRC-broadcast builds link memory using control messages that acknowledges the delivery of preceding messages. Every process safely removes obsolete control information about past broadcast messages. The size of the local structure increases and decreases over receipts. In the next section, we analyze the complexity of this causal broadcast implementation.

2.5 Complexity

In this section, we analyze the complexity of PRC-broadcast in terms of broadcast message overhead, delivery execution time, local space consumption, and number of control messages.
The **broadcast message overhead** is constant $O(1)$. The protocol uses reliable FIFO links to transmit messages.
The **delivery execution time**, i.e., the time complexity of the receipt function is $O(|Q_i|)$. The protocol checks and updates control information associated to each link in the in-view Q_i. The size of in-views can be much smaller than the number of processes in the system $|P|$. For instance, peer-sampling approaches [8, 21] provides every process with an in-view the size of which is logarithmically scaling with the number of processes $O(\ln(|P|))$.
The **local space consumption** depends on the size of buffers and the size of the in-view. Each link in the in-view has its buffer of control information about messages. A message appears in the structure after its first receipt and disappears at its last receipt. So the local space complexity is $O(|Q_i| \cdot M)$ where M is the number of messages already delivered that will be received again from at least a link in the in-view Q_i. The local space consumption depends on system settings (e.g. processes do not consume space when the system topology is a ring or a tree) and use (e.g. processes do not consume space when no process broadcasts any message).
The overhead in terms of **number of control messages** per added link in an out-view varies from 6 to $4 \cdot |P|^2$ depending on the overlay network; P being the set of processes currently

in the system. It achieves 6 messages when Process A adds Process B using Process C as mediator, and Process B has Process A in its out-view. It achieves 8 control messages when peer-sampling protocols build out-views using neighbor-to-neighbor interactions [11, 21]. It achieves $O(4 \cdot \log(|P|))$ control messages when peer-sampling protocols allows processes to route their messages [12, 25]. It achieves $O(4 \cdot |P|^2)$ control messages when Process A adds Process B without knowledge of any route. Process A and Process B fall back to reliable broadcast instead of routing to disseminate control messages.

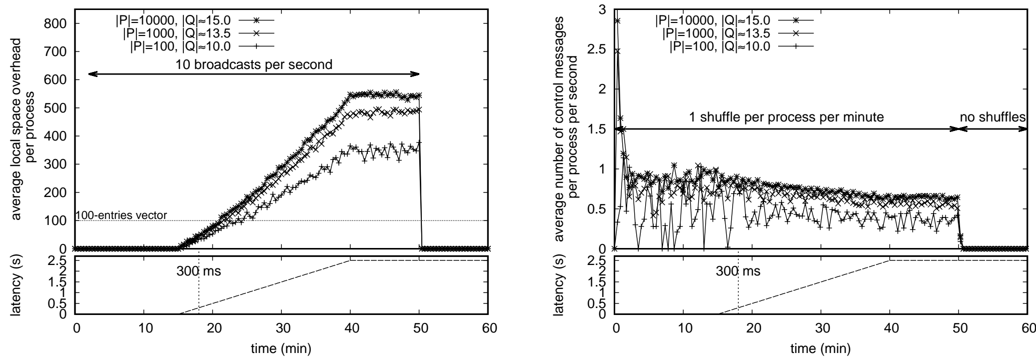
This complexity analysis shows that PRC-broadcast proposes a novel trade-off in terms of complexity. In systems allowing a form of routing, processes only send a few control messages to handle dynamicity. Every process maintains a local structure the size of which increases and decreases over receipts. Every process safely removes obsolete control information about past messages. It constitutes an advantageous trade-off that depends on the actual system settings and use instead of past deliveries. The next section describes an experiment highlighting the effects of the system settings on the space consumed by processes.

3 Experimentation

PRC-broadcast proposes a novel trade-off between speed, memory, and traffic. Most importantly, its space consumed varies over receipts. In this section, we evaluate the impact of the actual system on the space consumed and traffic generated by processes. The experiments run on the PeerSim simulator [17] that allows to build large and dynamic systems. Our implementation is available on the Github platform at `http://github.com/chat-wane/peersim-prcbroadcast`.

Objective: To confirm that local space complexity depends on in-views and message receipts.

Description: We measure the average size of buffers and arrays of expected messages. This constitutes the average local space overhead consumed by PRC-broadcast to detect and forbid multiple delivery in dynamic systems.
Runs involve 3 overlay networks comprising 100, 1k, and 10k processes. Spray [21] builds a highly dynamic overlay networks. The resulting topology has properties close to those of random graphs such as low diameter, or low clustering coefficient. Such systems are highly resilient to random crashes, and allows processes to balance the load of the traffic generated by broadcasting. Each process maintains an out-view logarithmically scaling with the number of processes in the system. Each process of the 100-processes system has an out-view of ≈ 10 neighbors. Each process of the 1k-processes system has an out-view of ≈ 13.5 neighbors. Each process of the 10k-processes system has a out-view of ≈ 15 neighbors. Each process dynamically reconfigures its out-view: it gives half of its correctly initialized links to a chosen neighbor; the latter gives half of its correctly initialized links to the former as well. Each exchange leads to link memory initialization and safety checks of the new links, and removal of given links. Each process starts to reconfigures its out-view as soon as it joins the system and reconfigures its out-view every minute. This uniformly spreads reconfigurations over the duration of the experiment.
Links are bidirectional, their safety must be checked in both directions but the overhead remains minor. Since the peer-sampling protocol that builds the system uses neighbor-to-neighbor communication to establish new links, each control message is two hops away from its destination. Overall, a new link requires 8 control messages to be initialized properly. Links have transmission delay, i.e., the time between the sending of a message and its receipt



(a) Local space overhead (number of control information about broadcast messages in all buffers).

(b) Generated traffic overhead (number of control messages transiting in the system including routed messages).

Figure 5 Overhead of PRC-broadcast required to ensure causal order and forbid multiple delivery in dynamic systems with varying latency.

is not null. The experiments start with 1 millisecond transmission delay. At 15 minutes, the delay starts to increase. At 17 minutes, links reach 300 milliseconds transmission delay. At 40 minutes, links reach 2.5 seconds transmission delay and it stops increasing.
From 2 minutes to 50 minutes, every second, 10 processes chosen uniformly at random among all processes broadcast a message.

Results: Figure 5a shows the results of this experiment. The x-axis denotes the time in minute. The top part of the figure shows the local space overhead while the bottom part of the figure shows the evolution of transmission delays.
Figure 5a confirms that the local space consumption depends on the in-view size. Systems with larger in-views consume more space. Each new delivered message adds control information on each link of the in-view (see Algorithm 1).
Figure 5a confirms that the local space consumption depends on network condition. The overhead increases as the latency increases. Latency increases the time between the first and the last receipt of each message. Processes store messages longer until their safe removal.
Figure 5a confirms that the local space consumption depends on broadcast messages. When processes stops broadcasting, the space consumed at each process drops to 0. Each process eventually receive each message and safely remove the corresponding entry.
Figure 5a shows that at a rate of 10 broadcasts per second and when latency stays under a realistic bound (300 milliseconds), the overhead is lower than vector-based approaches. Whatever system conditions, it would require a vector of 100, 1k entries, 10k entries to forbid multiple delivery in the 100-processes system, 1k-processes system, 10k-processes system respectively. However, it is worth noting that the overhead of PRC-broadcast increases linearly with the number of messages currently transiting. 100 broadcasts per second would multiply measurements made on PRC-broadcast by a factor of 10. In such case, the 100-entries vector would be better than PRC-broadcast even under a latency of 300 milliseconds.
PRC-broadcast provides a novel trade-off between speed, memory, and traffic. Among other, its space consumed increases and decreases depending on the system and its current use; instead of past use (see Section 4. This result means that it constitutes an advantageous trade-off in

(i) dynamic systems
(ii) comprising up to millions of processes
(iii) that could broadcast at any time.

Objective: To confirm that the generated traffic overhead depends on the dynamicity of the system.

Description: We measure the average number of control messages received by each process during a second. This includes the routing of messages. The setup is identical to that of prior experiment.

Results: Figure 5b shows the results of this experiment. The top part of the figure depicts the traffic overhead generated by PRC-broadcast while the bottom part of the figure depicts the evolution of transmission delays.
Figure 5b shows that the number of control messages received by processes depends on the dynamicity of the system. The more dynamic the higher the traffic overhead. At the beginning of the experiment, processes join the system. Numerous links are established at once, hence the high number of control messages. Then processes shuffle their out-view during 50 minutes. The number of links to add and remove is roughly constant over time, hence the stabilization in number of control messages. Finally, processes stop shuffling at 50 minutes. Processes do not receive additional control messages.
Figure 5b confirms our traffic overhead complexity analysis. For instance, in the 10k-processes system, views comprises 15 processes which belong half from the out-view and half from the in-view. Each process shuffles every minute. Each shuffle adds and removes 7.5 links (twice half of the out-view size). Since the peer-sampling protocol establishes links using neighbor-to-neighbor interactions, it allows a form of routing where only 8 control messages are required to initialize a new link. $|exchanged_links| * |control_messages|/60 \approx 7.5 * 8/60 \approx 1$ control message per second.
Figure 5b shows that latency smooth and decreases the number of control messages. The peer-sampling protocol only shuffles links already safe and the memory of which is initialized. Since increasing latency increases the initialization time of links, processes exchange less links at each shuffle. The generated traffic decreases accordingly. Latency also spreads control messages over time, hence the smoothing in measurements.
Assuming peer-sampling protocols that enable a form of routing, PRC-broadcast forbids multiple delivery at the cost of a few lightweight control messages in dynamic systems. In this experiment, the underlying peer-sampling protocol builds a random graph topology that has numerous desirable properties such as resilience to failures, quick dissemination of information, or load balancing [11]. It fits dynamic systems where numerous processes join and leave continuously. Nonetheless, other peer-sampling protocols could be used depending on the configuration of the system. One could minimize latency [5], or gather people based on user preferences [12].

Overall, this section showed that PRC-broadcast proposes a novel trade-off in terms of complexity. Its complexity actually depends on the system (its dynamicity, its latency, its topology) and current use (broadcasts per second). PRC-broadcast forbids multiple delivery and safely removes obsolete control information about broadcast messages. The next section reviews state-of-the-art approaches designed to forbid multiple delivery.

Table 1 Complexity of broadcast algorithms at each process. N the number of processes that ever broadcast a message. P is the set of processes in the system. W the number of messages received but not delivered yet. Q_i is the set of incoming links. M is the number of messages already delivered that will be received again from at least one link in Q_i.

	message overhead	delivery execution time	local space consumption	# control messages per added link
reliable broadcast [9]	$O(1)$	$O(1)$	$O(N)$	0
causal broadcast [23]	$O(N)$	$O(W \cdot N)$	$O(N + W \cdot N)$	0
preventive broadcast [20]	$O(1)$	$O(1)$	$O(N)$	3 to $2 \cdot \lvert P \rvert^2$
this paper	$\mathbf{O(1)}$	$\mathbf{O(\lvert Q_i \rvert)}$	$\mathbf{O(\lvert Q_i \rvert \cdot M)}$	**6** to $\mathbf{4 \cdot \lvert P \rvert^2}$

4 Related work

Causal broadcast ensures causal order and forbids multiple delivery. PRC-broadcast uses the former to improve on the complexity of the latter. This section reviews state-of-the-art broadcast protocols that forbid multiple delivery in asynchronous and dynamic systems.

Building **specific dissemination topologies** such as tree or ring guarantees that every process receives each message once [4, 22]. Processes deliver messages as soon as they arrive. They do not need to save any control information about messages, for they will never receive a copy of this message again. While these approaches are lightweight, they stay confined to systems where failures are uncommon, and where churn rate remains low [13]. PRC-broadcast generalizes on these specific topologies. It follows the same principle where the topology impacts on the number of receipts. Its space complexity scales linearly with this number of receipts. In turns, PRC-broadcast inherits from the resilience of the underlying topology maintained by processes. PRC-broadcast supports dynamic systems without assuming any specific topology.

Without any specific dissemination topology, each process may receive each broadcast message multiple times. Despite multiple receipts, a process must deliver a message once. Using local structures based on **logical clocks [14]**, every process differentiates between the first receipt of a broadcast message and the additional receipts of this message. It allows to deliver the former while ignoring the latter. Unfortunately, the size of these structures increases monotonically and linearly with the number of processes that ever broadcast a message [16, 18]. Processes cannot reclaim the space consumed, for it would require running an overcostly distributed garbage collection that is equivalent to a distributed consensus [1]. This limits their use to context where the number of broadcasters is known to be small. PRC-broadcast uses local structures based on logical clocks too. However, instead of saving the past deliveries of broadcasters, its saves the messages expected from direct neighbors. The set of expected messages varies over receipts, and the number of neighbors can be far smaller than the set of broadcasters. PRC-broadcast scales in large and dynamic systems. Among others, PRC-broadcast fits contexts where the number of participants is unknown, such as distributed collaborative editing [19].

Table 1 summarizes the complexity of broadcast implementations that handle asynchronous and dynamic systems. To the best of our knowledge, all causal broadcast implementations use an underlying reliable broadcast in order to forbid multiple delivery. Their local space complexity comprises $O(N)$ where N is the number of processes that ever broadcast a message. Compared to preventive causal broadcast [20], PRC-broadcast slightly increases the delivery execution time, and doubles the number of control message per added link. In turns, PRC-broadcast keeps a constant overhead on broadcast message, and changes the terms of

local space complexity. Most importantly, the local space consumed does not monotonically increase anymore.

5 Conclusion

In this paper, we proposed a causal broadcast implementation that provides a novel trade-off between speed, memory, and traffic. Our approach exploits causal order to improve on the space complexity of the implementation that forbids multiple delivery. The local space complexity of this protocol does not monotonically increase and depends at each moment on the number of messages still in transit and the degree of the communication graph. The overhead in terms of number of control messages depends on the dynamicity of the system and remains low upon the assumption that the overlay network allows a form of routing. This advantageous trade-off makes causal broadcast a lightweight and efficient middleware for group communication in distributed systems.

As future work, we plan to investigate on ways to retrieve the partial order of messages out of PRC-broadcast. Applications may require more than causal order, they also may need to identify concurrent messages [26]. PRC-broadcast discards a lot of information by ignoring multiple receipts altogether. Analyzing the receipt order could provide insight on the partial order. The cost could depend on the actual concurrency of the system.

References

1. Saleh E. Abdullahi and Graem A. Ringwood. Garbage Collecting the Internet: A Survey of Distributed Garbage Collection. *ACM Comput. Surv.*, 30(3):330–373, September 1998.
2. Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 761–772, New York, NY, USA, 2013. ACM.
3. Dhruba Borthakur. Petabyte Scale Databases and Storage Systems at Facebook. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1267–1268, New York, NY, USA, 2013. ACM.
4. Manuel Bravo, Luís Rodrigues, and Peter Van Roy. Saturn: A Distributed Metadata Service for Causal Consistency. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 111–126, New York, NY, USA, 2017. ACM.
5. Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: A Decentralized Network Coordinate System. *SIGCOMM Comput. Commun. Rev.*, 34(4):15–26, 2004.
6. Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12. ACM, 1987.
7. Roy Friedman and Shiri Manor. Causal Ordering in Deterministic Overlay Networks. *Israel Institute of Technology: Haifa, Israel*, 2004.
8. Ayalvadi J. Ganesh, Anne-Marie Kermarrec, and Laurent Massoulié. Scamp: Peer-to-Peer Lightweight Membership Service for Large-Scale Group Communication. In Jon Crowcroft and Markus Hofmann, editors, *Networked Group Communication*, volume 2233 of *Lecture Notes in Computer Science*, pages 44–55. Springer Berlin Heidelberg, 2001.
9. Vassos Hadzilacos and Sam Toueg. A Modular Approach to Fault-Tolerant Broadcasts and Related Problems. Technical report, Cornell University, Ithaca, NY, USA, 1994.
10. Matthias Heinrich, Franz Lehmann, Thomas Springer, and Martin Gaedke. Exploiting single-user web applications for shared editing: a generic transformation approach. In *Proceedings of the 21st international conference on World Wide Web*, pages 1057–1066. ACM, 2012.

11 Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten Van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems (TOCS)*, 25(3):8, 2007.

12 Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. T-Man: Gossip-based fast overlay topology construction. *Computer Networks*, 53(13):2321–2339, 2009.

13 Sveta Krasikova, Raziel C. Gómez, Heverson B. Ribeiro, Etienne Rivière, and Valerio Schiavoni. Evaluating the Cost and Robustness of Self-organizing Distributed Hash Tables. In *Proceedings of the 16th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems - Volume 9687*, pages 16–31, Berlin, Heidelberg, 2016. Springer-Verlag.

14 Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

15 Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

16 Dahlia Malkhi and Doug Terry. Concise version vectors in WinFS. *Distributed Computing*, 20(3):209–219, 2007.

17 Alberto Montresor and Márk Jelasity. PeerSim: A Scalable P2P Simulator. In *Proceedings of the 9th International Conference on Peer-to-Peer (P2P'09)*, pages 99–100, Seattle, WA, September 2009.

18 Madhavan Mukund, Gautham Shenoy R., and S.P. Suresh. Optimized OR-Sets without Ordering Constraints. In Mainak Chatterjee, Jian-nong Cao, Kishore Kothapalli, and Sergio Rajsbaum, editors, *Distributed Computing and Networking*, volume 8314 of *Lecture Notes in Computer Science*, pages 227–241. Springer Berlin Heidelberg, 2014.

19 Brice Nédelec, Pascal Molli, and Achour Mostéfaoui. CRATE: Writing Stories Together with Our Browsers. In *Proceedings of the 25th International Conference Companion on World Wide Web*, WWW '16 Companion, pages 231–234, Republic and Canton of Geneva, Switzerland, 2016. International World Wide Web Conferences Steering Committee.

20 Brice Nédelec, Pascal Molli, and Achour Mostéfaoui. Breaking the Scalability Barrier of Causal Broadcast for Large and Dynamic Systems. In *Proceedings of the 37th IEEE International Symposium on Reliable Distributed Systems*, SRDS '18. IEEE, 2018.

21 Brice Nédelec, Julian Tanke, Davide Frey, Pascal Molli, and Achour Mostéfaoui. An adaptive peer-sampling protocol for building networks of browsers. *World Wide Web*, August 2017.

22 Michel Raynal. *Distributed algorithms for message-passing systems*, volume 500. Springer, 2013.

23 Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174, March 1994.

24 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Rapport de recherche RR-7506, INRIA, January 2011.

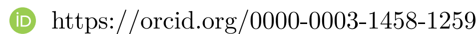
25 Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160, New York, USA, 2001. ACM.

26 David Sun and Chengzheng Sun. Context-Based Operational Transformation in Distributed Collaborative Editing Systems. *IEEE Transactions on Parallel and Distributed Systems*, 20(10):1454–1470, October 2009.

Output-Oblivious Stochastic Chemical Reaction Networks

Ben Chugg[1]
The University of British Columbia, Canada
ben.ih.chugg@gmail.com

Hooman Hashemi
The University of British Columbia, Canada
hhoomn390@gmail.com

Anne Condon[2]
The University of British Columbia, Canada
condon@cs.ubc.ca

https://orcid.org/0000-0003-1458-1259

Abstract

We classify the functions $f : \mathbb{N}^2 \to \mathbb{N}$ which are stably computable by *output-oblivious* Stochastic Chemical Reaction Networks (CRNs), i.e., systems of reactions in which output species are never reactants. While it is known that precisely the semilinear functions are stably computable by CRNs, such CRNs sometimes rely on initially producing too many output species, and then consuming the excess in order to reach a correct stable state. These CRNs may be difficult to integrate into larger systems: if the output of a CRN $\mathcal{C}$ becomes the input to a downstream CRN $\mathcal{C}'$, then $\mathcal{C}'$ could inadvertently consume too many outputs before $\mathcal{C}$ stabilizes. If, on the other hand, $\mathcal{C}$ is output-oblivious then $\mathcal{C}'$ may consume $\mathcal{C}$'s output as soon as it is available. In this work we prove that a semilinear function $f : \mathbb{N}^2 \to \mathbb{N}$ is stably computable by an output-oblivious CRN with a leader if and only if it is both increasing and either *grid-affine* (intuitively, its domains are congruence classes), or the minimum of a finite set of *fissure functions* (intuitively, functions behaving like the min function).

2012 ACM Subject Classification Theory of computation → Computability, Theory of computation → Formal languages and automata theory

Keywords and phrases Chemical Reaction Networks, Stable Function Computation, Output-Oblivious, Output-Monotonic

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.21

Related Version A full version of the paper is available at `https://arxiv.org/abs/1812.04401`.

1 Introduction

Stochastic Chemical Reaction Networks (CRNs) – systems of reactions involving chemical species – have traditionally been used to reason about extant physical systems, but are currently also of strong interest as a distributed computing model for describing molecular programs [8, 16]. They are closely related to Population Protocols [1, 3, 4, 9], another

[1] Supported by an NSERC Undergraduate Student Research Award.
[2] Supported by an NSERC Discovery Grant.

© Ben Chugg, Hooman Hashemi, and Anne Condon;
licensed under Creative Commons License CC-BY
22nd International Conference on Principles of Distributed Systems (OPODIS 2018).
Editors: Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira; Article No. 21; pp. 21:1–21:16

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

distributed computing model; these models have found applications in areas as diverse as signal processing [13], graphical models [14], neural networks [12], and modeling cellular processes [5, 6]. CRNs can simulate Universal Turing Machines [16, 2]. However, these simulations have drawbacks: the number of reactions or molecules may scale with the space usage and the computation is only correct with an arbitrarily small probability of error. If we require *stable computation* – that the CRN always eventually produces the correct answer – then Angluin et al. [4] showed that precisely the class of semilinear predicates can be stably computed. Chen et al. [8] extended this result to show that precisely the semilinear functions can be stably computed.

Recent advances in physical implementations of CRNs and, more generally, chemical computation using strand displacement systems (e.g., [15, 17, 18, 19]) are a step towards the use of CRNs in biological environments and nanotechnology. As these systems become more complex, it may be necessary to integrate multiple, interacting CRNs in one system. However, current CRN constructions may perform poorly in such scenarios. As a concrete example, consider a CRN $\mathcal{C}$ given by the reactions $X \to 2Y$, $Y + L \to \emptyset$, where the system begins with n copies of input species X, and one copy of L (called the *leader*). This CRN eventually produces $2n - 1$ copies of output species Y, and so (stably) computes the function $n \mapsto 2n - 1$. If another CRN $\mathcal{C}'$ uses the output of $\mathcal{C}$ as its input, and if the first reaction occurs n times before the second occurs at all, then $\mathcal{C}'$ may consume all $2n$ copies of Y and may thus itself produce an erroneous output. Current CRN constructions circumvent this issue by using *diff-representation*, where the count y of output species Y of a CRN is represented indirectly as the difference $y = y^P - y^C$ between the counts of two species Y^P and Y^C [8], rather than as the count of one output species Y. While these constructions enable the counts of both Y^C and Y^P to be non-decreasing throughout the computation, it is not immediately clear how a second CRN might use these two species reliably as input.

More generally, if multiple function-computing CRNs comprise a larger system it can be desirable that no CRN ever produces a number of outputs that exceeds its function value. We might even demand more: that an output species of a CRN is never used as a reactant species, i.e., is never consumed. This ensures that any secondary CRN relying on the first's output can consume the output indiscriminately.

It is thus natural to ask: What functions can be stably computed in an *output-oblivious manner*, in which outputs are never reactants, without using diff-representation?

This question is the focus of this paper. Doty and Hajiaghayi [11] already observed that output-oblivious functions must not only be semilinear but also increasing, that is, $f(\mathbf{n}_1) \leq f(\mathbf{n}_2)$ whenever $\mathbf{n}_1 \leq \mathbf{n}_2$, but did not provide further insights. Chalk et al. [7] asked the same question but for a different model, namely mass-action CRNs. That model tracks real-valued species concentrations, unlike the stochastic model in which configurations are vectors of species counts. In contrast with the mass-action mode, leader molecules can play a very important role in the stochastic model, and we focus on the case where leaders are present. Mass-action CRN models cannot have leaders since there are no species counts. Functions that are stably computable by output-oblivious mass-action CRNs must be super-additive [7], that is $f(n) + f(n') \leq f(n + n')$. Semilinear functions that are super-additive are a proper subset of the class of output-oblivious functions (characterized in this paper) that can be stably computed by stochastic CRNs with leaders.

1.1 Our Results

In this work we characterize the class of output-oblivious semilinear functions, i.e., those functions that can be stably computed by an output-oblivious stochastic CRN. We assume that one copy of a leader species is present initially in addition to the input. We focus on

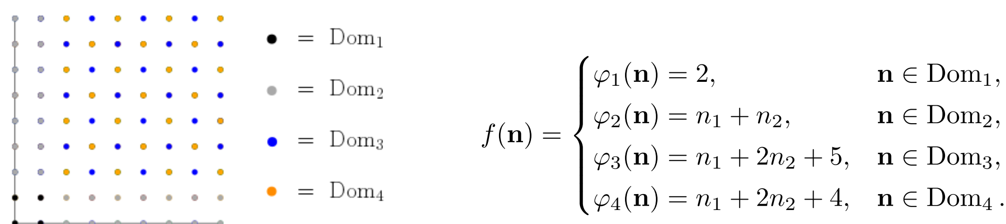


Figure 1 Here we represent a grid-affine function $f : \mathbb{N}^2 \to \mathbb{N}$ by its decomposition on different domains, all of which are grids. The domains of f are illustrated on the left. Each black point is a zero-dimensional grid, while the grey points represent four one-dimensional grids, namely the lines $\{(\alpha, 0) + (2, i) : \alpha \in \mathbb{N}\}$ and $\{(0, \alpha) + (i, 2) : \alpha \in \mathbb{N}\}$ for $i = 0, 1$. The blue points represent points (n_1, n_2) such that $n_1 + n_2$ is even, and cover the union of two grids: $\{(2\alpha_1, 2\alpha_2) : \alpha_i \in \mathbb{N}\} \cup \{(2\alpha_1, 2\alpha_2) + (1, 1) : \alpha_i \in \mathbb{N}\}$. Similarly, the gold points represent two grids.

functions with two inputs and one output, since this case already is quite complex. Our results generalize trivially when there are more outputs since each output can be handled independently, and we believe that our techniques also generalize to multiple inputs.

Our results also hold for Population Protocols, since stable function-computing CRNs can be translated into Population Protocols and vice versa. Section 2 introduces the relevant background in order to formally describe our results, but we describe them informally here.

Perhaps the simplest type of output-oblivious function with domain $\mathbb{N}^2$ is an affine function, such as $f(n_1, n_2) = 2n_1 + 3n_2 + 1$ which could be computed by a CRN with reactions $L \to Y$, $X_1 \to 2Y$ and $X_2 \to 3Y$ where L is a single leader. Here and hereafter, X_i will typically correspond to the input species representing n_i.

In Section 3 we show that an increasing function that can be specified as partial affine functions whose domains are different "grids" of $\mathbb{N}^2$ is also output-oblivious; for example, the function $f(n_1, n_2) = 2n_1 + 3n_2 + 1$ when $n_1 + n_2 = 0 \mod 2$, and $f(n_1, n_2) = 2n_1 + 3n_2$ when $n_1 + n_2 = 1 \mod 2$. More generally, a function that can be specified in terms of output-oblivious partial functions f_i, $1 \leq i \leq k$, defined on different grids of $\mathbb{N}^2$, is output-oblivious. The grids may be 0-dimensional, in which case they are points; 1-dimensional in which case they are lines, or 2-dimensional. We call such functions *grid-affine* functions. See Figure 1 for a slightly more complicated example of a grid-affine function, and a representation of its domains. We show how the CRNs for partial functions f_i on the different grids can be "stitched" together to obtain an output-oblivious CRN for f.

It is also straightforward to obtain an output-oblivious CRN for a function f that is the min of a finite set of output-oblivious functions. In the simplest case, for example, $\min(n_1, n_2)$ can be computed as $X_1 + X_2 \to Y$. In our main positive result we describe a more general type of "min-like" function, which we call a *fissure function*, and we show how to construct output-oblivious CRNs for such functions. We give a very simple example of a fissure function and a corresponding output-oblivious CRN in Figure 2.

However, constructing CRNs for other fissure functions appears to be significantly trickier than that shown in Figure 2. Consider the function $f(n_1, n_2) = 2n_1 + 3n_2 + 2$ if $n_1 > n_2$, $f(n_1, n_2) = 3n_1 + 2n_2 + 2$ if $n_1 < n_2$ and $f(n_1, n_2) = 5n_1$ on the "fissure line" $n_1 = n_2$. The simple line-tracking mechanism of the CRN of Figure 2 can't be used here because the affine functions for the "wedge" domains "$n_1 > n_2$" and "$n_1 < n_2$" depend both on n_1 and n_2. Also the function cannot be written as the sum of an increasing grid-affine function and an increasing simple fissure function of the type in Figure 2, where the "above" function $\varphi_A()$ depends only on n_1 and the "below" function $\varphi_B()$ depends only on n_2. Our main positive result is a construction that can handle such fissure functions, as well as functions

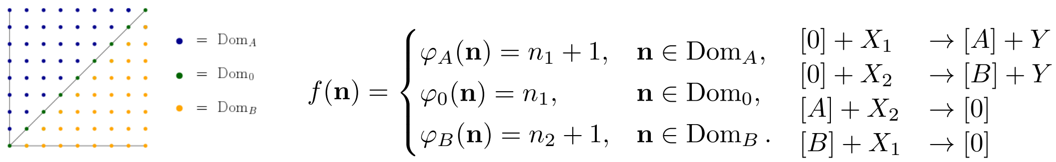


Figure 2 A simple fissure function $f : \mathbb{N}^2 \to \mathbb{N}$. On the left the three domains of f are illustrated. There is one "fissure line" called Dom_0, and two "wedge" domains called Dom_A and Dom_B ("A" is above and "B" is below the fissure line). The function value on each of these domains is specified in the center. The function f agrees with the function $\min\{n_1 + 1, n_2 + 1\}$ except that it dips down by 1 on the fissure line Dom_0. On the right is a CRN which stably computes f. In the CRN, the input $\mathbf{n} = (n_1, n_2)$ is represented as counts of species X_1 and X_2 and the leader is initially $[0]$. The three possible states $[0], [A]$ and $[B]$ of the leader track whether the input lies on the fissure line Dom_0, which is the line where $\varphi_A(\mathbf{n}) - \varphi_B(\mathbf{n}) = 0$, or whether the input lies above or below the fissure line, i.e., in domains Dom_A or Dom_B respectively. In this simple example, the CRN need not track how far above (or how far below) the fissure line an input might be, since the function φ_A does not depend on n_2 (and the function φ_B does not depend on n_1).

with multiple parallel fissure lines.

In Section 4 we present results on the negative side. A non-trivial example of a function that is not output-oblivious is the maximum function. Intuitively, a CRN that attempts to compute the max would have to keep track of the relative difference of its two inputs in order to know when the count of one input overtakes the count of the other, and it's not possible to keep track of that difference with a finite number of states. Developing this intuition further, we show that an increasing semilinear function $f : \mathbb{N}^2 \to \mathbb{N}$ is output-oblivious if and only if f is grid-affine or is the min of finitely many fissure functions.

Putting both positive and negative results together, we state our main result here (see Section 2 for precise definitions of grid-affine and fissure functions).

▶ **Theorem 1.** *A semilinear function $f : \mathbb{N}^2 \to \mathbb{N}$ is output-oblivious if and only if f is increasing and is either grid-affine or the minimum of finitely many fissure functions.*

Since only semilinear functions are stably computable by CRNs, Theorem 1 provides a complete characterization of functions $f : \mathbb{N}^2 \to \mathbb{N}$ which are output-oblivious. Moreover, in Section 4, we will prove that if f is output-*monotonic*, then it is either grid-affine or the minimum of fissure functions, a stronger statement than in Theorem 1. A function is output-monotonic if it is stably computable by a CRN whose output count never decreases but unlike an output-oblivious CRN an output may act as a catalyst of a reaction, being both a reactant and product. For example, the CRN $X \to Y$, $L + Y \to 2Y$ which computes the function $n \mapsto n + 1$ for $n \geq 1$ and $0 \mapsto 0$ is output-monotonic, but not output-oblivious. Thus, we also obtain a characterization for output-monotonic functions.

To obtain our results, we provide new characterizations of semilinear sets and functions. We show that all semilinear sets can be written as finite unions of sets which are the intersection of grids and hyperplanes. Such sets are points, lines or wedges (pie-shaped slices) on 2D grids. Using this and the representation of semilinear functions as piecewise affine functions discovered by Chen et al. [8], we give a new representation of semilinear functions as "periodic semiaffine functions", essentially piecewise affine functions whose domains are points, lines or wedges.

The rest of the paper is structured as follows. Section 2 provides the relevant technical background on CRNs, stable computation and semilinear functions. It also contains our new results on the structure of semilinear sets and functions, and rigorous definitions of grid-affine and fissure functions. In the remaining two sections we prove Theorem 1, with Section 3

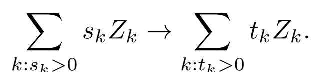
providing explicit constructions of CRNs and Section 4 proving that any function which is stably computable by an output-oblivious CRN obeys certain properties. Some proofs are omitted and will appear in the full version of the paper.

2 Preliminaries

We begin by introducing Chemical Reaction Networks, and what it means for a CRN to stably compute a function. We then formally define grid-affine and fissure functions and, along the way, state new results concerning semilinear sets and functions.

2.1 Chemical Reaction Networks (CRNs)

CRNs specify possible behaviours of systems of interacting *species*. Let $\mathcal{Z} = \{Z_1, \ldots, Z_m\}$ be a finite set of species. At any given instant, the system is described by a configuration $\mathbf{c} \in \mathbb{N}^{\mathcal{Z}}$, where $c(Z_i)$ is the current count of the species $Z_i \in \mathcal{Z}$ in the system. The system's configuration changes by way of *reactions*, each of which is described as a pair $(\mathbf{s}, \mathbf{t}) = ((s_1, \ldots, s_m), (t_1, \ldots, t_m)) \in \mathbb{N}^{\mathcal{Z}} \times \mathbb{N}^{\mathcal{Z}}$ such that for at least one $1 \leq j \leq m$, $s_j \neq t_j$. Reaction $(\mathbf{s}, \mathbf{t})$ can be written as



$$\sum_{k: s_k > 0} s_k Z_k \to \sum_{k: t_k > 0} t_k Z_k.$$

The species Z_k with $s_k > 0$ are the *reactants*, which are *consumed*, while those with $t_k > 0$ are the *products* (if both $s_k > 0$ and $t_k > 0$ then species Z_k is a catalyst). A CRN is thus formally described as a pair $\mathcal{C} = (\mathcal{Z}, \mathcal{R})$, where $\mathcal{Z}$ is a set of species, and $\mathcal{R}$ a set of reactions. Reaction $r = (\mathbf{s}, \mathbf{t})$ is *applicable* to configuration $\mathbf{c}$ if $\mathbf{s} \leq \mathbf{c}$ (pointwise inequality), i.e., sufficiently many copies of each reactant are present. If applicable reaction $(\mathbf{s}, \mathbf{t})$ occurs when the system is in configuration $\mathbf{c} = (c_1, \ldots, c_m)$, the new configuration is $\mathbf{c}' = (c_1 - s_1 + t_1, \ldots, c_m - s_m + t_m)$. In this case we say that $\mathbf{c}'$ is *directly reachable* from $\mathbf{c}$ and write $\mathbf{c} \xrightarrow{r} \mathbf{c}'$. An *execution* $\mathcal{E} = \mathbf{c}_0, \ldots, \mathbf{c}_t$ of $\mathcal{C}$ is a sequence of configurations of $\mathcal{C}$ such that $\mathbf{c}_i$ is directly reachable from $\mathbf{c}_{i-1}$ for $1 \leq i \leq t$. We say that $\mathbf{c}_t$ is *reachable* from $\mathbf{c}_0$.

Stable CRN Computation of functions with a leader. Angluin et al. [3] introduced the concept of stable computation of boolean predicates by population protocols, and Chen et al. [8] adapted the notion to function computation by CRNs. While this paper focuses on two-dimensional domains, we present the following details in full generality.

Let $f : \mathbb{N}^k \to \mathbb{N}^\ell$ be a function. Formally, a *Chemical Reaction Network (CRN) for computing f with a leader* is $\mathcal{C} = (\mathcal{Z}, \mathcal{R}, \mathcal{I}, \mathcal{O}, L)$, where $\mathcal{Z}$ is a set of species, $\mathcal{R}$ is a set of reactions, $\mathcal{I} = \{X_1, X_2, \ldots, X_k\} \subseteq \mathcal{Z}$ is an ordered set of input species, $\mathcal{O} = \{Y_1, Y_2, \ldots, Y_\ell\} \subseteq \mathcal{Z}$ is an ordered set of output species and L is a leader species, $L \in \mathcal{Z} \setminus \mathcal{I}$.

Function computation on input $\mathbf{n} = (n_1, \ldots, n_k) \in \mathbb{N}^k$ starts from a *valid initial configuration* $\mathbf{c}_0$ of $\mathcal{C}$; namely a configuration in which the count of L is 1, the count of species X_i is n_i, and the count of any other species is 0. A *computation* is an execution of $\mathcal{C}$ from a valid initial configuration to a stable configuration. A configuration c is *stable* if for every $\mathbf{c}' \in \mathbb{N}^m$ reachable from $\mathbf{c}$, $\mathbf{c}(Y) = \mathbf{c}'(Y)$ for all $Y \in \mathcal{O}$. That is, once the system reaches configuration $\mathbf{c}$, the counts of the output species do not change. We say that $\mathcal{C}$ *stably computes* f if for every valid initial configuration $\mathbf{c}_0$ and for every configuration $\mathbf{c}$ reachable from $\mathbf{c}_0$, there exists a stable configuration $\mathbf{c}'$ reachable from $\mathbf{c}$ such that $f(\mathbf{c}_0(X_1), \ldots, \mathbf{c}_0(X_k)) = (\mathbf{c}'(Y_1), \ldots, \mathbf{c}'(Y_\ell))$.

Output-monotonic and output-oblivious CRNs. We say a CRN $\mathcal{C}$ is *output-oblivious* if it never consumes any of its output species, and *output-monotonic* if on all executions from a valid initial configuration, the count of any output species never decreases. As noted in the introduction, these notions are not equivalent. We say a function f is *output-oblivious* (*monotonic*) if there exists an output-oblivious (monotonic) CRN which stably computes f. Our results show that the set of output-oblivious functions and output-monotonic functions are the same.

2.2 Linear and Semilinear Sets; Lines, Grids, and Wedges

For a vector $\mathbf{v}$, let v_i denote its ith coordinate. Let $D \subseteq \mathbb{N}^2$ and let Π_1 and Π_2 denote the projection maps onto x and y axes, respectively. We say D is *two-way-infinite* if $|\Pi_1(D)| = |\Pi_2(D)| = \infty$, *one-way-infinite* if either $|\Pi_1(D)| = \infty$ or $|\Pi_2(D)| = \infty$ but not both, and finite if $|\Pi_1(D)| < \infty$ and $|\Pi_2(D)| < \infty$. Also, if $A, B \subseteq \mathbb{N}^2$ and $\mathbf{n} \in \mathbb{N}^2$ we let $A + B = \{a + b : a \in A, b \in B\}$ and $A + \mathbf{n} = A + \{\mathbf{n}\}$.

A set $E \subset \mathbb{N}^2$ is *linear* if $E = \{\sum_{i=1}^{t} \mathbf{x}_i \alpha_i + \mathbf{o} : \alpha_i \in \mathbb{N}\}$ for some $t \in \mathbb{N}$ and $\mathbf{x}_i, \mathbf{o} \in \mathbb{N}^2$. If $t = 1$ we say that E is a *line*. A set is *semilinear* if it is the finite union of linear sets.

A linear set $\mathcal{G} \subseteq \mathbb{N}^2$ is a *grid* if there exist $p, q \in \mathbb{N}$ and $\mathbf{o} \in \mathbb{N}^2$ such that $\mathcal{G} = \{(p, 0)\alpha_1 + (0, q)\alpha_2 : \alpha_i \in \mathbb{N}\} + \mathbf{o} = \{(p\alpha_1 + o_1, q\alpha_2 + o_2) : \alpha_i \in \mathbb{N}\}$. If both p and q are zero, the grid is simply the point $\mathbf{o}$. If $p > 0$ and $q = 0$, or $p = 0$ and $q > 0$, the grid is a one-way-infinite line with period p or q respectively. If $p = q > 0$ we say that the grid is periodic, with period p. We let $\mathcal{G}_p + \mathbf{o}$ be the grid $\{(\alpha_1 p, \alpha_2 p) : \alpha_i \in \mathbb{N}\} + \mathbf{o}$ and write $\mathcal{G}_p$ if $\mathbf{o} = (0, 0)$.

A *threshold set* is a linear set with the form $\{\mathbf{x} : \mathbf{x} \cdot \mathbf{v} \geq r\}$ (i.e., a halfspace) for some $\mathbf{v} \in \mathbb{Z}^2$ and $r \in \mathbb{Z}$. Let E be a two-way-infinite linear set of the form $\mathcal{G} \cap \mathcal{T}$, where $\mathcal{G}$ is a grid and $\mathcal{T}$ is a finite intersection of threshold sets. E is bounded by two lines (represented by threshold sets and/or the x or y axes; the points on these lines, if any, are in E). If the two bounding lines are parallel, E is the finite union of lines on $\mathcal{G}$, i.e., all points of each line lie on grid $\mathcal{G}$. Otherwise we call E a *wedge* on $\mathcal{G}$. For example, the sets $\{\mathbf{n} : n_1 \geq n_2\}$ and $\{(1, 1)\alpha_1 + (1, 2)\alpha_2 : \alpha_i \in \mathbb{N}\}$ are wedges on $\mathcal{G}_1$. Likewise, the two regions above and below the fissure line in Figure 2 are wedges on $\mathcal{G}_1$. More generally, we can intuitively think of a wedge as a pie-like slice of $\mathbb{N}^2 \cap \mathcal{G}$, except that pieces may be chopped off near the narrow "corner" that is closest to the origin. If the two bounding lines are the x and y axes, the wedge is all of $\mathcal{G}$. We can show the following characterization of semilinear sets.

▶ **Lemma 2.** *Every semilinear set can be represented as the finite union of points, lines on grids, and wedges on grids, with all grids having the same period.*

2.3 Semilinear, Semiaffine, Grid-Affine, and Fissure Functions

For a function $f : \mathbb{N}^2 \to \mathbb{N}$, the *restriction of f to domain $D \subseteq \mathbb{N}^2$* is the partial function $f|_D : D \to \mathbb{N}$ given by $f|_D(\mathbf{n}) = f(\mathbf{n})$ for all $\mathbf{n} \in D$. We say that $f : D \to \mathbb{N}$ is *(partial) affine* if $f(\mathbf{n}) = a_1 n_1 + a_2 n_2 + a_0$ for rational numbers a_0, a_1, and $a_2 \in \mathbb{Q}$. Function f is a *finite combination* of the finite set of functions $\{\varphi_1, \ldots, \varphi_k\}$ if $\mathrm{Dom}(f) = \bigcup_{i=1}^{k} \mathrm{Dom}(\varphi_i)$ and $f(\mathbf{n}) = \varphi_i(\mathbf{n})$ whenever $\mathbf{n} \in \mathrm{Dom}(\varphi_i)$. Throughout we write Dom_i in place of $\mathrm{Dom}(\varphi_i)$. We define semilinear functions using a characterization of Chen et al. [8]:

▶ **Definition 3** (Semilinear function [8])**.** A function $f : \mathbb{N}^2 \to \mathbb{N}$ is *semilinear* if and only if f is a finite combination of partial affine functions with linear domains.

We next define *semiaffine* functions, a refinement of Definition 3. Lemma 5 then states that semilinear and semiaffine functions are equivalent.

▶ **Definition 4** (Semiaffine function). Let $\mathcal{G}_p+\mathbf{o}$ be a periodic grid. A function $f : \mathcal{G}_p+\mathbf{o} \to \mathbb{N}$ is *semiaffine* if and only if f is a finite combination of partial affine functions whose domains are points, lines or wedges on grid $\mathcal{G}_p+\mathbf{o}$. A function $f : \mathbb{N}^2 \to \mathbb{N}$ is *semiaffine with period* $p \in \mathbb{N}^+$ if and only if f is a combination of semiaffine functions on grids of the form $\mathcal{G}_p+\mathbf{o}$.

▶ **Lemma 5.** *A function $f : \mathbb{N}^2 \to \mathbb{N}$ is semilinear if and only if f is semiaffine.*

Our main result, Theorem 1, shows that output-oblivious functions are exactly the following two special types of semiaffine functions. In the first special case, on each grid $\mathcal{G}_p+\mathbf{o}$, f is restricted to be an affine (rather than a more general semiaffine) function.

▶ **Definition 6** (Grid-affine function). A function $f : \mathbb{N}^2 \to \mathbb{N}$ is *grid-affine* if and only if for some $p \in \mathbb{N}^+$, f is a combination of affine functions on points and on grids of period p.

A function $f : D \to \mathbb{N}$ is *increasing* if $f(\mathbf{n}) \leq f(\mathbf{n}')$ for all $\mathbf{n} \leq \mathbf{n}'$, where $\mathbf{n}, \mathbf{n}' \in D$. Doty and Hajiaghayi [11] observed that an output-oblivious function must be increasing. Accordingly, we hereafter focus on increasing functions.

▶ **Definition 7** (Fissure function). Let $\mathcal{G}$ be a two-way-infinite grid. An increasing semiaffine function $f : \mathcal{G} \to \mathbb{N}$ is a *partial fissure function* if for some $\mathbf{o} \in \mathbb{N}^2$, f can be represented as follows for all $\mathbf{n} \geq \mathbf{o}$:

$$f(\mathbf{n}) = \begin{cases} \varphi_A(\mathbf{n}), & \text{if } \varphi_A(\mathbf{n}) - \varphi_B(\mathbf{n}) \leq -k, \\ \varphi_{-i}(\mathbf{n}) = \varphi_A(\mathbf{n}) - d_{-i}, & \text{if } \varphi_A(\mathbf{n}) - \varphi_B(\mathbf{n}) = -i, 1 \leq i < k, \\ \varphi_i(\mathbf{n}) = \varphi_B(\mathbf{n}) - d_i, & \text{if } \varphi_A(\mathbf{n}) - \varphi_B(\mathbf{n}) = i, 0 \leq i < k, \\ \varphi_B(\mathbf{n}), & \text{if } \varphi_A(\mathbf{n}) - \varphi_B(\mathbf{n}) \geq k. \end{cases} \tag{1}$$

where $\varphi_A(\mathbf{n}) = A_0 + A_1 n_1 + A_2 n_2$, $\varphi_B(\mathbf{n}) = B_0 + B_1 n_1 + B_2 n_2$, for integers A_0 and B_0, nonnegative rationals A_1, A_2, B_1 and B_2, and nonnegative integers $d_{-k}, \ldots, d_{-1}, d_0, d_1, \ldots, d_k$. For $-k \leq i \leq k$, we refer to the line $\varphi_A(\mathbf{n}) - \varphi_B(\mathbf{n}) = i$ as a *fissure line* and call it L_i. Moreover, $\varphi_A < \varphi_B$ on Dom_A and $\varphi_B < \varphi_A$ on Dom_B; thus $A_1 > B_1$ and $B_2 > A_2$. We say $f : \mathbb{N}^2 \to \mathbb{N}$ is a *(complete) fissure function* if f is a combination of partial fissure functions on grids of period p.

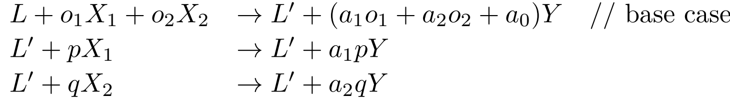
3 Proof of Sufficiency in Theorem 1

This section shows that if an increasing semilinear function $f : \mathbb{N}^2 \to \mathbb{N}$ is either a grid-affine function or a fissure function, then f is output-oblivious. We do this in three lemmas. Lemma 8 shows that an increasing affine function whose domain is a grid is output-oblivious. Lemma 10 shows that a partial fissure function is output-oblivious. Finally, Lemma 11 shows that if f is increasing and is a combination of partial output-oblivious functions defined on grids, we can stitch together the CRNs for the partial functions to obtain an output-oblivious CRN for f.

▶ **Lemma 8.** *Let $\mathcal{G}$ be a grid. Any increasing affine function $f : \mathcal{G} \to \mathbb{N}$ is output-oblivious.*

Proof. We consider the case that $\mathcal{G} = \{(p,0)\alpha_1 + (0,q)\alpha_2 : \alpha_i \in \mathbb{N}\} + \mathbf{o}$ is two-way-infinite; the cases when $\mathcal{G}$ is a point or a line are simpler. Let $f(\mathbf{n}) = a_1 n_1 + a_2 n_2 + a_0$, where $a_1, a_2 \in \mathbb{Q}^+$ and $a_0 \in \mathbb{Q}$. Since $\mathcal{G}$ is two-way-infinite and f is increasing, a_1 and a_2 are

nonnegative. On input $\mathbf{n} = (n_1, n_2) \in \mathcal{G}$, i.e., given n_1 copies of X_1 and n_2 copies of X_2, the following CRN will produce $f(\mathbf{n})$ copies of Y:



$$\begin{aligned} L + o_1X_1 + o_2X_2 &\to L' + (a_1o_1 + a_2o_2 + a_0)Y \quad \text{// base case} \\ L' + pX_1 &\to L' + a_1pY \\ L' + qX_2 &\to L' + a_2qY \end{aligned}$$

Note that the first reaction must produce a non-negative and integral number of Y's since $f(\mathbf{o}) \in \mathbb{N}$. Likewise, $a_1p \in \mathbb{N}$ since $a_1p = f(\mathbf{o} + (p, 0)) - f(\mathbf{o})$, and similarly for a_2q. Finally, the CRN is clearly output-oblivious since the output species Y is never a reactant. ◀

We show in Lemma 10 below that any partial fissure function is output-oblivious. First we describe some useful structure pertaining to partial fissure functions $f : \mathcal{G} \to \mathbb{N}$. We can represent such a fissure function as $f(\mathbf{n}) = \min\{\varphi_A(\mathbf{n}), \varphi_B(\mathbf{n})\} - d_i$, where d_i is determined by the fissure line L_i on which $\mathbf{n}$ resides, and $d_i = 0$ if i is not on a fissure line; this formulation is not identical to but is equivalent to that of Definition 7. As noted in that definition, it must be that $A_1 > B_1$ and $B_2 > A_2$, since $\varphi_A < \varphi_B$ on Dom_A and vice versa.

For all integers i, let L_i be the line $\varphi_A(\mathbf{n}) - \varphi_B(\mathbf{n}) = i$. All of these lines, which include the $2k - 1$ "fissure lines" $L_i, -k < i < k$, have the same slope. In addition to the fissure lines, our CRN construction will also refer to the lines L_i for i in the range $[k, \ldots, K - 1]$, where $K = k + d_{\max} - 1$. We call these the *lower boundary lines*, and we call the lines L_i for i in the range $[-K + 1, \ldots, -k]$ the *upper boundary lines*. Note that $(0, 0)$ is on the line $L_{A_0 - B_0}$ and more generally, if point $\mathbf{p}$ is on line L_i then $(A_1 - B_1)p_1 - (B_2 - A_2)p_2 = i - A_0 + B_0$. For $\mathbf{n} \in \mathcal{G}$ let $M(\mathbf{n}) = (\varphi_A(\mathbf{n}), \varphi_B(\mathbf{n}))$. The next lemma shows that $M(\mathbf{n}) \in \mathbb{N}^2$ for all sufficiently large $\mathbf{n} \in \mathcal{G}$, even though Dom_A and Dom_B are proper subsets of $\mathcal{G}$.

▶ **Lemma 9.** *Let $\varphi : D \to \mathbb{N}$ be a partial affine function, where D is a wedge domain on $\mathcal{G}$. Let $\mathbf{m}$ be a minimal point of D. Then $\varphi(\mathbf{n}) \in \mathbb{N}$ on all $\mathbf{n} \in \mathcal{G}$ with $\mathbf{n} \geq \mathbf{m}$.*

We let $\mathcal{P}$ be the set of rational points $\mathbf{p}$ for which $M(\mathbf{p}) \in \mathbb{N}$ and let $\mathcal{Q}$ be the range of M with respect to domain $\mathcal{P}$. For $\mathbf{q} \in \mathcal{Q}$, let $M^{-1}(\mathbf{q})$ denote the inverse of M ($M^{-1}\mathbf{q}$ is unique since (A_1, A_2) and (B_1, B_2) are linearly independent). The following claim follows easily from the definition of M and will be useful later.

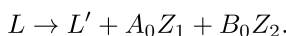
▶ **Claim 1.** *Let $z_1, z_1', z_2, z_2' \in \mathbb{N}$. If $z_1 \leq z_1'$ and $M^{-1}(z_1, z_2)$ is in Dom_B then $M^{-1}(z_1', z_2)$ is also in Dom_B. Similarly if $z_2 \leq z_2'$ and $M^{-1}(z_1, z_2)$ is in Dom_A then $M^{-1}(z_1, z_2')$ is also in Dom_A.*

▶ **Lemma 10.** *Any partial fissure function $f : \mathcal{G} \to \mathbb{N}$ is output-oblivious.*

Proof. For simplicity we assume that the grid $\mathcal{G}$ is $\mathbb{N}^2$, i.e., the period of the grid is 1 and the offset $\mathbf{o}$ is zero; it is straightforward to generalize to larger grid periods. With these assumptions, it must be that A_0 and B_0 are nonnegative integers, which slightly simplifies base cases of our construction.

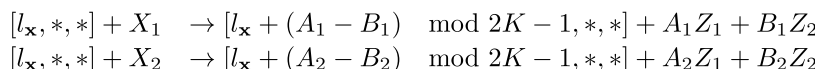
The CRN input is represented as the initial counts of species X_1 and X_2, and $\mathbf{x} = (x_1, x_2)$ denotes the counts of X_1 and X_2 that have been consumed at any time. Rather than producing output $f(\mathbf{x})$ directly upon consumption of $\mathbf{x}$, our CRN produces $\varphi_A(\mathbf{x})$ copies of a species Z_1 and $\varphi_B(\mathbf{x})$ copies of a species Z_2, effectively computing the mapping M described above. Note that $\varphi_A(\mathbf{x})$ and $\varphi_B(\mathbf{x})$ are nonnegative integers by Lemma 9. The CRN works backwards from the quantities $\varphi_A(\mathbf{x})$ and $\varphi_B(\mathbf{x})$ to reconstruct $f(\mathbf{x})$. Roughly

this is possible because $f(\mathbf{x})$ is "almost" the min of $\varphi_A(\mathbf{x})$ and $\varphi_B(\mathbf{x})$, and min is easy to compute. More precisely, we can assume that $f(\mathbf{x}) = \min\{\varphi_A(\mathbf{x}), \varphi_B(\mathbf{x})\} - d_i$, where d_i is determined by the fissure line L_i on which $\mathbf{n}$ resides, and $d_i = 0$ if i is not on a fissure line. In addition to the input, a leader L is also present initially. Other CRN molecules (not initially present) represent a state $[l_\mathbf{x}, l_\mathbf{z}, d]$ containing three components; we explain the components later. Our CRN has three types of reactions: *Z-producing*, *Z-consuming*, and *Y-producing* reactions. The first Z-producing reaction handles the base case, producing $(\varphi_A(0,0), \varphi_B(0,0)) = (A_0, B_0)$:



$$L \to L' + A_0 Z_1 + B_0 Z_2.$$

The remaining two Z-producing reactions consume X_1 and X_2 while producing Z_1 and Z_2. If L_i is the line containing $\mathbf{x}$, the first state component, $l_\mathbf{x}$, keeps track of $i \mod 2K-1$, where $K = k + d_{\max}$. If i is in the range $[-K+1, K-1]$ then $l_\mathbf{x}$ uniquely determines i. For convenience in what follows, we consider $l_\mathbf{x}$ to be in the range $[-K+1, K-1]$ rather than $[0, 2K-1]$. The reactions are as follows, where $*$ represents any state component value that is unchanged as a result of the reaction:

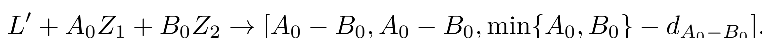


$$\begin{aligned} [l_\mathbf{x}, *, *] + X_1 &\to [l_\mathbf{x} + (A_1 - B_1) \mod 2K-1, *, *] + A_1 Z_1 + B_1 Z_2 \\ [l_\mathbf{x}, *, *] + X_2 &\to [l_\mathbf{x} + (A_2 - B_2) \mod 2K-1, *, *] + A_2 Z_1 + B_2 Z_2 \end{aligned}$$

We next describe the Z-consuming reactions. These reactions update the remaining two components of the state to keep track of which fissure or boundary line contains $M^{-1}(\mathbf{z})$, where $\mathbf{z} = (z_1, z_2)$ denotes the counts of (Z_1, Z_2) that have been consumed at any time. The reactions also track what is the *deficit*, i.e., the difference between the "true" output $f(M^{-1}(\mathbf{z}))$ and the current output y, i.e., number of copies of species Y that has been actually produced so far. Formally, all reactions maintain the following *state invariant*: if after any reaction the state is $[l_\mathbf{x}, l_\mathbf{z}, d]$ then

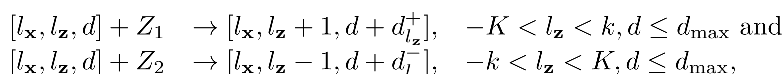
1. $l_\mathbf{z}$ is the index of the boundary or fissure line $L_{l_\mathbf{z}}$ that contains $M^{-1}(\mathbf{z})$, and $l_\mathbf{z}$ is in the range $-K+1 \leq l_\mathbf{z} \leq K-1$; and
2. $d = f(M^{-1}(\mathbf{z})) - y$ is the deficit in the number of y's produced, and is in the finite range $-d_{\max} \leq d \leq 2d_{\max} + 1$, where $d_{\max} = \max\{d_i \mid -k < i < k\}$.

Z-consuming reactions of the first type handle the base case when $\mathbf{n} = (0,0)$:



$$L' + A_0 Z_1 + B_0 Z_2 \to [A_0 - B_0, A_0 - B_0, \min\{A_0, B_0\} - d_{A_0 - B_0}].$$

Z-consuming reactions of the second type consume a copy of Z_1 and reactions of the third type consume a copy of Z_2. Upon consumption, the state components are updated to ensure that the state invariant holds.



$$\begin{aligned} [l_\mathbf{x}, l_\mathbf{z}, d] + Z_1 &\to [l_\mathbf{x}, l_\mathbf{z} + 1, d + d^+_{l_\mathbf{z}}], \quad -K < l_\mathbf{z} < k, d \leq d_{\max} \text{ and} \\ [l_\mathbf{x}, l_\mathbf{z}, d] + Z_2 &\to [l_\mathbf{x}, l_\mathbf{z} - 1, d + d^-_{l_\mathbf{z}}], \quad -k < l_\mathbf{z} < K, d \leq d_{\max}, \end{aligned}$$

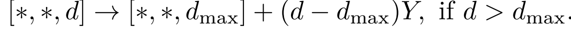
where

$$d^+_{l_\mathbf{z}} = \begin{cases} d_{l_\mathbf{z}} - d_{l_\mathbf{z}+1}, & l_\mathbf{z} \geq 0 \\ d_{l_\mathbf{z}} - d_{l_\mathbf{z}+1} + 1, & l_\mathbf{z} < 0. \end{cases} \quad \text{and } d^-_{l_\mathbf{z}} = \begin{cases} d_{l_\mathbf{z}} - d_{l_\mathbf{z}-1} + 1, & l_\mathbf{z} \geq 0 \\ d_{l_\mathbf{z}} - d_{l_\mathbf{z}-1}, & l_\mathbf{z} < 0. \end{cases}$$

The deficit d can never exceed $2d_{\max}+1$ since the reactions are only applicable when $d \leq d_{\max}$ and d can increase by at most $d_{\max} + 1$.

The Y-producing reactions produce output molecules of species Y, while maintaining the state invariant above, and ensuring that at the end of the computation the number of Ys

produced equals $f(\mathbf{n})$. The first Y-producing reaction produces $d - d_{\max}$ copies of Y when d becomes greater than $d_{\max}$.



$$[*, *, d] \to [*, *, d_{\max}] + (d - d_{\max})Y, \text{ if } d > d_{\max}.$$

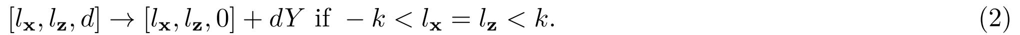
Before describing the remaining Y-producing reactions, we describe some properties of the system of reactions above. We say that Z-consumption *stalls* if none of the Z-consuming reactions are ever applicable again. Let $\mathbf{z}_s = (z_{1s}, z_{2s})$ be the counts of (Z_1, Z_2) consumed when Z-consumption stalls ($\mathbf{z}_s$ is independent of the order in which the reactions happen). The Y-producing reaction above ensures that the Z-consuming reactions are never stalled because d becomes too large. Also, the Z-consuming reactions don't stall if $l_{\mathbf{z}}$ is a fissure line and another Z_1 is or will eventually be available (and similarly if another Z_2 is or will eventually be available), because ł_z changes by 1 upon consumption of Z_1 and so is still less than K.

Stalling happens when and only when one of the following (exclusive) cases arise. (i) All copies of both Z_1 and Z_2 have been consumed and no more will ever be produced, so $\mathbf{z}_s = (\varphi_A(\mathbf{n}), \varphi_B(\mathbf{n}))$. (ii) All copies of Z_2 have been consumed and no more will ever be produced, so $z_{2s} = \varphi_B(\mathbf{n})$ but $z_{1s} < \varphi_A(\mathbf{n})$. In this case, $M^{-1}(\mathbf{z}_s)$ is on a lower boundary line. To see why, note that if $M^{-1}(\mathbf{z}_s)$ were on a fissure or upper boundary line, then the Z-consuming reaction that consumes Z_1 would eventually be applicable, because $l_{\mathbf{z}}$ is in the proper range and at least one copy of Z_1 has yet to be consumed. (iii) All copies of Z_1 have been consumed and no more will ever be produced, so $z_{1s} = \varphi_A(\mathbf{n})$, but $z_{2s} < \varphi_B(\mathbf{n})$. In this case, the line $L_{l_{\mathbf{z}}}$ containing $M^{-1}(\mathbf{z}_s)$ must be an upper boundary line.

▶ **Claim 2.** $f(M^{-1}(\mathbf{z}_s)) = f(\mathbf{n})$.

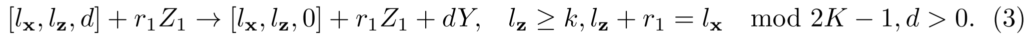
Proof. This is trivial in case (i) when all Zs have been consumed and no more will be produced, since $M^{-1}(\mathbf{z}_s) = \mathbf{n}$. Consider case (ii) (case (iii) is similar). Then $M^{-1}(\mathbf{z}_s) = M^{-1}(z_{1s}, \varphi_B(n))$, $z_{1s} < \varphi_A(\mathbf{n})$, and the line containing $M^{-1}(\mathbf{z}_s)$ is a lower boundary line. By Claim 1, $\mathbf{n}$ must be in Dom_B, because $\mathbf{n} = M^{-1}(\varphi_A(\mathbf{n}), \varphi_B(\mathbf{n}))$ and $\varphi_A(\mathbf{n}) > z_{1s}$. Therefore, $f(M^{-1}(\mathbf{z}_s)) = \varphi_B(M^{-1}(\mathbf{z}_s)) = \varphi_B(\mathbf{n}) = f(\mathbf{n})$. ◀

We now return to the last three reactions of the CRN, which are Y-producing reactions; we will number them so that we can reference them later and refer to them as *deficit-clearing* reactions. The next reaction clears a positive deficit when both $\mathbf{x}$ and $M^{-1}(\mathbf{z})$ lie on the same fissure line:



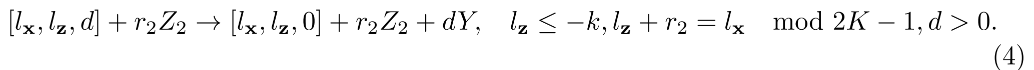
$$[l_{\mathbf{x}}, l_{\mathbf{z}}, d] \to [l_{\mathbf{x}}, l_{\mathbf{z}}, 0] + dY \text{ if } -k < l_{\mathbf{x}} = l_{\mathbf{z}} < k. \tag{2}$$

The last two Y-producing reactions clear the deficit if $M^{-1}(\mathbf{z})$ is on a lower boundary line and for some nonnegative integer r, $M^{-1}(\mathbf{z} + (r, 0))$ is on a line L_l with $l = l_{\mathbf{x}} \mod 2K - 1$. If such an r exists, let r_1 be the smallest such integer and add the following reaction:



$$[l_{\mathbf{x}}, l_{\mathbf{z}}, d] + r_1 Z_1 \to [l_{\mathbf{x}}, l_{\mathbf{z}}, 0] + r_1 Z_1 + dY, \quad l_{\mathbf{z}} \geq k, l_{\mathbf{z}} + r_1 = l_{\mathbf{x}} \mod 2K - 1, d > 0. \tag{3}$$

We add a similar reaction when the line $L_{l_{\mathbf{z}}}$ containing $M^{-1}(\mathbf{z})$ is an upper boundary line, when an similarly-defined r_2 exists:



$$[l_{\mathbf{x}}, l_{\mathbf{z}}, d] + r_2 Z_2 \to [l_{\mathbf{x}}, l_{\mathbf{z}}, 0] + r_2 Z_2 + dY, \quad l_{\mathbf{z}} \leq -k, l_{\mathbf{z}} + r_2 = l_{\mathbf{x}} \mod 2K - 1, d > 0. \tag{4}$$

This completes the description of the CRN. We need one more claim in order to complete the proof of the lemma:

▶ **Claim 3.** *When Z-consumption stalls, the deficit is nonnegative.*

To complete the proof, we argue that once Z-consumption stalls, some deficit-clearing reaction will eventually be applicable, ensuring that the output eventually produced is $f(\mathbf{n})$. If $M^{-1}(\mathbf{z}_s)$ is on a fissure line then $M^{-1}(\mathbf{z}_s)$ must equal $\mathbf{n}$, in which case Y-producing reaction (2) is applicable. If $M^{-1}(\mathbf{z}_s)$ is on a boundary line then either (3) or (4) will be applicable once all inputs are consumed, since for some r, either $M^{-1}(\mathbf{z}_s + (r, 0)) = \mathbf{n}$ or $M^{-1}(\mathbf{z}_s + (0, r)) = \mathbf{n}$. Thus in all cases some Y-producing reaction eventually clears the deficit, ensuring that the output produced is $f(\mathbf{n})$. ◀

▶ **Lemma 11.** *(Stitching Lemma) Let Let f be an increasing function. If $f : \mathbb{N}^2 \to \mathbb{N}$ is a finite combination of output-oblivious functions whose domains are grids, then f is output-oblivious. Also if f is the min of a finite number of output-oblivious functions then f is output-oblivious.*

Proof Sketch. Let f be a finite combination of output-oblivious functions, say $f_1, f_2, \ldots, f_m$, whose domains are grids. We first describe the construction for the case that the domain Dom_j of f_j is a two-way-infinite grid for all $j, 1 \leq j \leq m$. Let the offset of the jth grid be $\mathbf{o}_j = (o_{j,1}, o_{j,2})$. On input $\mathbf{n}$, our CRN $\mathcal{C}$ first produces m distinct "inputs" $\mathbf{n}^{(\mathbf{j})} \in \mathbb{N}^2$ such that $\mathbf{n} \leq \mathbf{n}^{(\mathbf{j})}$ and $\mathbf{n} = \mathbf{n}^{(\mathbf{j})}$ if $\mathbf{n} \in \text{Dom}_j$. From these, $\mathcal{C}$ produces m "outputs" $y_j = f_j(\mathbf{n}^{(\mathbf{j})})$, using CRNs $\mathcal{C}_j$ for each f_j. Finally, $\mathcal{C}$ produces $y = \min\{y_1, \ldots, y_m\}$.

To see that such a $\mathcal{C}$ is correct, i.e., that $y = f(\mathbf{n})$, note that if $\mathbf{n} \in \text{Dom}_j$ then $y_j = f_j(\mathbf{n}^{(\mathbf{j})}) = f_j(\mathbf{n}) = f(\mathbf{n})$, since $\mathbf{n} = \mathbf{n}^{(\mathbf{j})}$, and if $\mathbf{n} \notin \text{Dom}_j$ then $y_j = f_j(\mathbf{n}^{(\mathbf{j})}) \geq f(\mathbf{n})$, since f is increasing and $\mathbf{n}^{(\mathbf{j})} \geq \mathbf{n}$. Thus $f(\mathbf{n}) = \min\{y_1, \ldots, y_m\} = y$. The details of producing the $\mathbf{n}^{(\mathbf{j})}$s and the output are found in the full version.

When f is the min of a finite number of output-oblivious functions, say $f_1, f_2, \ldots, f_m$, we can similarly stably compute each f_i using an output-oblivious CRN $\mathcal{C}_i$ such that the species for each $\mathcal{C}_i$ are distinct, and then take the min of the outputs as the result. ◀

We complete this section by proving the sufficiency (if) direction of Theorem 1.

Theorem 1 (if direction). *A semilinear function $f : \mathbb{N}^2 \to \mathbb{N}$ is output-oblivious if f is increasing and is either grid-affine or the minimum of finitely many fissure functions.*

Proof. First suppose that f is grid-affine. Then by Definition 6, f is a combination of affine functions $f_1, \ldots, f_m$ whose domains $\mathcal{G}_1, \ldots, \mathcal{G}_m$ respectively are grids. By Lemma 8, $f_i : \mathcal{G}_i \to \mathbb{N}$ is output-oblivious, $1 \leq i \leq m$. By Lemma 11, f is output-oblivious.

Otherwise f is the min of finitely many complete fissure functions $f_1, \ldots, f_m$. By Definition 7, each f_i is a combination of partial fissure functions on grids of period p, for some $p \in \mathbb{N}$. By Lemma 10, each of these partial fissure functions is output-oblivious. By Lemma 11, each f_i is output-oblivious and also f is output-oblivious. ◀

4 Proof of Necessity in Theorem 1

In this section we prove that if a function is output-monotonic, then it is either grid-affine or the minimum of finitely many fissure functions. In Section 4.1 we describe two conditions on a function which ensure that it is not output-oblivious. In Section 4.2 we show two technical results which are needed in Section 4.3, which contains the proof of necessity. The arguments made in this section pertain to output-monotonic functions, allowing us to both characterize this set of functions and output-oblivious functions since any output-oblivious function is clearly output-monotonic.

4.1 Impossibility Lemmas

The results of this section use Dickson's Lemma:

▶ **Lemma 12.** *(Dickson's Lemma [10]) Any infinite sequence in $\mathbb{N}^k$ has an infinite, non-decreasing subsequence.*

▶ **Lemma 13.** *Let $f : \mathbb{N}^2 \to \mathbb{N}$ be a semiaffine function. Suppose that $f = \varphi_i$ on Dom_i and $f = \varphi_j$ on Dom_j, where Dom_i and Dom_j lie on the same grid, Dom_i is a wedge domain and for some two-way-infinite line L in Dom_j, $\varphi_j > \varphi_i$ on L. Then f cannot be stably computed by an output-monotonic CRN.*

Proof. Suppose to the contrary that CRN $\mathcal{C}$ stably computes f. Either Dom_i is counter-clockwise to Dom_j or vice versa. Assume it is the latter; the proof is similar if the orientation of the domains is reversed.

Let $\{\mathbf{p}_k\}_{k\in\mathbb{N}}$, be an infinite sequence of points in Dom_i that are strictly increasing in the first dimension, and form a line which is not parallel to L (this is possible since Dom_i is a wedge domain). Let $\mathbf{c}_k$ be a stable configuration reached on a computation of $\mathcal{C}$ on input $\mathbf{p}_k$. Applying Dickson's Lemma, choose an infinite subsequence of $\{\mathbf{c}_k\}$ and renumber so that $\mathbf{c}_k \le \mathbf{c}_{k+1}$ for all k. Let $\mathbf{n}_1, \mathbf{n}_2, \ldots$ be another strictly increasing sequence in $\mathbb{N}^2$ such that $\mathbf{p}_k + \mathbf{n}_k \in L$ and $\mathbf{p}_{k'} + \mathbf{n}_k \in \mathrm{Dom}_i$ for $k' > k$. Such a sequence exists because $\{\mathbf{p}_k\}$ is not parallel to L, and L is two-way-infinite.

If $\mathcal{C}$ is correct, then on input $\mathbf{p}_k + \mathbf{n}_k$ some execution sequence of $\mathcal{C}$ first reaches configuration $\mathbf{c}_k$, which has $\varphi_i(\mathbf{p}_k)$ copies of Y, and then outputs $\varphi_j(\mathbf{p}_k + \mathbf{n}_k) - \varphi_i(\mathbf{p}_k)$ additional Ys (since $\mathbf{p}_k + \mathbf{n}_k \in \mathrm{Dom}_j$). Let $k' > k$. On input $\mathbf{p}_{k'} + \mathbf{n}_k$ (which is in Dom_i), $\mathcal{C}$ can output a number of Ys equal to:

$$\varphi_i(\mathbf{p}_{k'}) + \varphi_j(\mathbf{p}_k + \mathbf{n}_k) - \varphi_i(\mathbf{p}_k).$$

This occurs when $\mathcal{C}$ first produces stable output $\varphi_i(\mathbf{p}_{k'})$ while consuming input $\mathbf{p}_{k'}$ and reaching configuration $\mathbf{c}_{k'}$. Since $\mathbf{c}_{k'} \ge \mathbf{c}_k$, it can then follow the same execution that it would follow from $\mathbf{c}_k$ to produce $\varphi_j(\mathbf{p}_k + \mathbf{n}_k) - \varphi_i(\mathbf{p}_k)$ copies of Y (since all the necessary species are available). However, since $\varphi_j > \varphi_i$ on L, the number of Ys produced is greater than

$$\varphi_i(\mathbf{p}_{k'}) + \varphi_i(\mathbf{p}_k + \mathbf{n}_k) - \varphi_i(\mathbf{p}_k) = \varphi_i(\mathbf{p}_{k'} + \mathbf{n}_k)$$

(the equality here follows since φ_i is affine). Thus too many Y's can be output by C on input $\mathbf{p}_{k'} + \mathbf{n}_k$, and so C cannot be output-monotonic. ◀

▶ **Lemma 14.** *Let $f : \mathbb{N}^2 \to \mathbb{N}$ be a semiaffine function. Suppose that $f = \varphi_i$ on Dom_i and $f = \varphi_j$ on Dom_j, where Dom_i and Dom_j are wedge domains on the same grid $\mathcal{G}$ such that (i) $\varphi_i = \varphi_j$ on $\mathbb{N}^2$ and (ii) there is a two-way-infinite line $L \subset \mathcal{G}$ separating Dom_i and Dom_j, with $\varphi_L < \varphi_i$ $(= \varphi_j)$ on L. Then f cannot be stably computed by an output-monotonic CRN.*

4.2 Properties of Increasing Semiaffine Functions

Here we show several useful properties of increasing semiaffine functions $f : \mathbb{N}^2 \to \mathbb{N}$. For a partial affine function φ_i with domain Dom_i, write $\varphi_i(\mathbf{n}) = \langle \mathbf{a}_i, \mathbf{n} \rangle + a_{0,i}$ (where $\langle \cdot, \cdot \rangle$ denotes the standard inner product). We say that a line L is a *constant distance* from Dom_i if there exists some constant K such that for all $\mathbf{n} \in L$ there is some $\mathbf{c}(\mathbf{n})$ with $-(K, K) \le \mathbf{c}(\mathbf{n}) \le (K, K)$ and $\mathbf{n} + \mathbf{c}(\mathbf{n}) \in \mathrm{Dom}_i$.

▶ **Lemma 15.** *Let $f : \mathbb{N}^2 \to \mathbb{N}$ be an increasing semiaffine function. Suppose that $f = \varphi_i$ on Dom_i and $f = \varphi_j$ on Dom_j, where φ_i and φ_j are partial affine functions, Dom_i and Dom_j are two-way infinite domains, and some line L in Dom_j is a constant distance from Dom_i. Then there exists some $\kappa \in \mathbb{Q}$ such that $\langle \mathbf{a}_i, \mathbf{n}\rangle = \langle \mathbf{a}_j, \mathbf{n}\rangle + \kappa$ for all $\mathbf{n} \in L$.*

▶ **Lemma 16.** *Let $f : \mathbb{N}^2 \to \mathbb{N}$ be an increasing semiaffine function. Suppose that $f = \varphi_i$ on Dom_i and $f = \varphi_j$ on Dom_j, where φ_i and φ_j are partial affine functions. If there exist two non-parallel lines $I, L \subset \mathrm{Dom}_i$ which are both a constant distance from Dom_j, then $\mathbf{a}_i = \mathbf{a}_j$.*

What follows is an easy consequence of the previous lemma.

▶ **Lemma 17.** *Let $f : \mathbb{N}^2 \to \mathbb{N}$ be an increasing semiaffine function. If there exists a two-way-infinite domain D on which φ_i and φ_j are well-defined such that $f(\mathbf{n}) = \varphi_i(\mathbf{n}) = \varphi_j(\mathbf{n})$ on all $\mathbf{n} \in D$, then $\varphi_i = \varphi_j$.*

4.3 Proof of Necessity

We first provide the main argument in the proof of necessity – that the desired result holds on individual grids.

▶ **Lemma 18.** *Let $f : \mathbb{N}^2 \to \mathbb{N}$ be an increasing semiaffine function with period p. Then f is output-monotonic only if for any large enough offset $\mathbf{o}$, $f|_{\mathcal{G}_p+\mathbf{o}}$ is either affine or the minimum of finitely many partial fissure functions.*

Proof. Fix a representation of f as a semiaffine function with period p. Choose $\mathbf{o}$ large enough so that, if $\mathcal{G} = \mathcal{G}_p + \mathbf{o}$, no domains of f that are points or one-way-infinite domains overlap $\mathcal{G}$, and also no two-way-infinite domains of f cross. Assume that $f|_\mathcal{G}$ is not affine; we need to show that $f|_\mathcal{G}$ is the minimum of partial fissure functions. Assume that the representation of f on $\mathcal{G}$ minimizes the number of wedge domains.

Consider all line domains in the representation of $f|_\mathcal{G}$, plus all two-way-infinite lines that define the top or bottom boundaries of wedge domains. Partition these lines into maximal sets of parallel lines. For each such set s, we define a function $f_s : \mathcal{G} \to \mathbb{N}^2$, and show in Claim 4 that each f_s is a partial fissure function. Let $\mathcal{S}$ be the set of all of the sets s of parallel lines. We show in Claim 5 that $f|_\mathcal{G} = \min_{s\in\mathcal{S}} f_s$, completing the proof of Lemma 18.

Definition of f_s. Fix any maximal set s of parallel lines. Without loss of generality we assume that s has at least three lines (we can add additional domains to f's representation if needed to ensure this). Some line of s, say $L_{A,s}$, defines the lower boundary of a wedge domain; we assume that this is top line of s (we can remove line domains of s above $L_{A,s}$ from f's representation if needed to ensure this). Let $f = \varphi_{A,s}$ on this wedge domain, where $\varphi_{A,s}$ is a partial affine function. In what follows, we drop the subscript s when referring to these and other domains and functions. Let Dom_A be the wedge of points of $\mathcal{G}$ that lie on or above L_A. (Note that Dom_A may not be a domain of $f|_\mathcal{G}$ since lines from some other set s_j may lie above the lines of s.) Similarly, we can assume that the bottom line, say L_B, of s defines the upper boundary of a wedge domain. Let $f = \varphi_B (= \varphi_{B,s})$ on this wedge domain and let Dom_B be the wedge of points of $\mathcal{G}$ that lie on or below L_B. We can assume without loss of generality (by adding more lines if necessary and further adjusting which lines are L_A and L_B) that any line L that lies between, and is parallel to, L_A and L_B is a (possibly empty) domain of $f|_G$. Number these lines $L_1, L_2, \ldots$, say from top to bottom, and let φ_i be the partial affine function that agrees with f on line L_i.

Let $f_s : \mathcal{G} \to \mathbb{N}$ be the following function associated with set s:

$$f_s(\mathbf{n}) = \begin{cases} \varphi_A(\mathbf{n}), & \text{if } \mathbf{n} \in \mathrm{Dom}_A, \\ \varphi_i(\mathbf{n}), & \text{if } \mathbf{n} \in L_i, 2 \leq i \leq |s| - 1, \\ \varphi_B(\mathbf{n}), & \text{if } \mathbf{n} \in \mathrm{Dom}_B\,. \end{cases}$$

▶ **Claim 4.** *f_s is a partial fissure function.*

Proof. Note that the lines L_i are parallel to, and have constant distance from, both L_A and L_B. By Lemma 15, for some $d_{i,A} \in \mathbb{Q}$ we have that $\varphi_i(\mathbf{n}) = \varphi_A(\mathbf{n}) - d_{i,A}$ for all $\mathbf{n}$ on line L_i. That is, L_i is the set of all $\mathbf{n}$ such that $\varphi_A(\mathbf{n}) - \varphi_i(\mathbf{n}) = d_{i,A}$. Since $\varphi_i(\mathbf{n})$ and $\varphi_A(\mathbf{n}) \in \mathbb{N}$ for all $\mathbf{n}$ in $\mathcal{G} \cap L_i$, $d_{i,A}$ must be in $\mathbb{Z}$. Also, since f is output-monotonic, Lemma 13 shows that $\varphi_i \leq \varphi_A$ on L_i and so $d_{i,A}$ must be in $\mathbb{N}$. By reasoning similar to that in the last paragraph, $\varphi_i(\mathbf{n}) = \varphi_B(\mathbf{n}) - d_{i,B}$ for all $\mathbf{n}$ on line L_i, for some $d_{i,B} \in \mathbb{N}$. It follows that L_i is the set of points $\mathbf{n} \in \mathcal{G}$ for which $\varphi_A(\mathbf{n}) - \varphi_B(\mathbf{n}) = k$, for some $k \in \mathbb{Z}$. It follows (by potentially adding yet more lines if necessary so that the number of lines L_i that lie above the line $\varphi_A(\mathbf{n}) - \varphi_B(\mathbf{n}) = 0$ is equal to the number of lines of s that lie below the line $\varphi_A(\mathbf{n}) - \varphi_B(\mathbf{n}) = 0$) that we can represent $f_s(\mathbf{n})$ as in Definition 7.

It remains to show that $\varphi_A < \varphi_B$ on Dom_A and $\varphi_B < \varphi_A$ on Dom_B. Suppose that there exists a point $\mathbf{n} \in \mathrm{Dom}_A$ such that $\varphi_A(\mathbf{n}) \geq \varphi_B(\mathbf{n})$. If there are only finitely many such points in any dimension, then we may disregard them by taking $\mathbf{o}$ sufficiently large. Hence, we may assume that if there is one such point then there are infinitely many and they form a two-way-infinite domain, D. Let $E = \{\mathbf{n} \in D : \varphi_A(\mathbf{n}) > \varphi_B(\mathbf{n})\}$ and $F = \{\mathbf{n} \in D : \varphi_A(\mathbf{n}) = \varphi_B(\mathbf{n})\}$. Note that $E \cup F = D$. We consider three cases. If E and F are both one-way-infinite, then one is a horizontal line and the other a vertical line. As above, we may disregard these points by taking $\mathbf{o}$ large enough. If E is two-way-infinite we can find a two-way-infinite line $L \subset \mathrm{Dom}_A$ such that $\varphi_A > \varphi_B$ on L. By Lemma 13, this contradicts the fact that f is output-monotonic. Otherwise, F is two-way-infinite. By Lemma 16 we see that $\varphi_A = \varphi_B$. Furthermore, for some i, we have $\varphi_i < \varphi_A$. Otherwise, $d_{i,A} = 0$ for all i and consequently the number of wedge domains of $f|_\mathcal{G}$ can be reduced by merging the domains Dom_A, Dom_B and the L_i contradicting our assumption that the representation of f on $\mathcal{G}$ minimizes the number of wedge domains. However, φ_A, φ_B and φ_i then meet the conditions of Lemma 14, a contradiction. The proof that $\varphi_B < \varphi_A$ on Dom_B is similar. ◀

▶ **Claim 5.** $f|_\mathcal{G} = \min_{s \in \mathcal{S}} f_s$.

Proof. Let $\mathbf{n} \in \mathrm{Dom}(f_s)$, so that $f|_\mathcal{G}(\mathbf{n}) = f_s(\mathbf{n})$. Suppose to the contrary that $f_{s'}(\mathbf{n}) < f_s(\mathbf{n})$ for some $s' \in \mathcal{S}$. We consider the case where the lines of s' lie above those of s; the case where the lines of s' lie below those of s is similar. Hence $f_{s'}(\mathbf{n}) = \varphi_{A,s'}(\mathbf{n})$. We consider three distinct cases based on the partitioning of $\mathrm{Dom}(f_s)$.

1. $\mathbf{n} \in \mathrm{Dom}_{A,s}$ so $f_s(\mathbf{n}) = \varphi_{A,s}(\mathbf{n})$. Applying the same argument as in the proof of Claim 4, since $\varphi_{A,s'}(\mathbf{n}) < \varphi_{A,s}(\mathbf{n})$, there is a line $L \subset \mathrm{Dom}_{A,s}$ such that $\varphi_{A,s'} < \varphi_{A,s}$ on L. Thus, $\varphi_{A,s'}$ and $\varphi_{A,s}$ and their respective domains meet the condition of Lemma 13, contradicting the fact that f can be stably computed by an output-monotonic CRN.
2. $\mathbf{n} \in \mathrm{Dom}_{i,s}$ for some i with $2 \leq i \leq |s| - 1$, so $f_s(\mathbf{n}) = \varphi_{i,s}(\mathbf{n})$. Since f_s is a fissure function, we have $\varphi_{i,s}(\mathbf{n}) = \varphi_{A,s}(\mathbf{n}) - d_i$, for some $d_i \in \mathbb{N}$, hence $f_s(\mathbf{n}) = \varphi_{i,s}(\mathbf{n}) \leq \varphi_{A,s}(\mathbf{n}) \leq \varphi_{A,s'}(\mathbf{n}) = f_{s'}(\mathbf{n})$. Clearly then, it cannot be the case that $f_{s'}(\mathbf{n}) < f_s(\mathbf{n})$.
3. $\mathbf{n} \in \mathrm{Dom}_{B,s}$. This is similar to Case 1. ◀

Since we get a contradiction in all three cases, we conclude that $f_s(\mathbf{n})$ must be less than or equal to $f_{s'}(\mathbf{n})$ for all $s' \in \mathcal{S}$, and the claim is proved.

This completes the proof of Lemma 18. ◀

Lemma 18 describes properties of increasing semiaffine functions on one grid. We now turn to properties of such functions across grids. The proof of the next lemma builds on that of Lemma 18 and uses Lemma 16 to relate f across grids.

▶ **Lemma 19.** *Let $f : \mathbb{N}^2 \to \mathbb{N}$ be an output-monotonic semiaffine function with period p. Let $\mathbf{o}$ be large enough that for all $\mathbf{o}' \geq \mathbf{o}$, $f|_{\mathcal{G}_p+\mathbf{o}'}$ is either affine or the minimum of finitely many partial fissure functions. If f is affine on $\mathcal{G} = \mathcal{G}_p + \mathbf{o}$, then $f|_{\mathcal{G}_p+\mathbf{o}'}$ must also be affine for all $\mathbf{o}' \geq \mathbf{o}$, and thus f is grid-affine. Otherwise f is the minimum of finitely many fissure functions.*

We complete this section by proving the necessity (only if) direction of Theorem 1, strengthening it slightly so that it applies also to output-monotonic functions.

Theorem 1 (only if direction). *A semilinear function $f : \mathbb{N}^2 \to \mathbb{N}$ is output-monotonic only if f is increasing and is either grid-affine or the minimum of finitely many fissure functions.*

Proof. By Lemma 5, f is a semiaffine function with period p. By Lemmas 18 and 19, f is either grid-affine or the minimum of fissure functions. ◀

5 Conclusions and Future Work

Here we have characterized the class of functions $f : \mathbb{N}^2 \to \mathbb{N}$ that can be stably computed by output-oblivious and output-monotonic stochastic chemical reaction networks (CRNs) with a leader. A natural next step for future work is to generalize the result to functions $f : \mathbb{N}^k \to \mathbb{N}$ for $k > 2$; we are optimistic that many of the building blocks that we introduce here for the two-dimensional case will generalize to the multi-dimensional case.

Another natural question is to determine what can be computed when there is no leader. By similar reasoning to that of Chalk et al. [7] for mass-action CRNs, such functions must be super-additive, but whether only the super-additive semilinear functions have output-oblivious stochastic CRNs remains to be determined.

Yet another direction for future work is to determine whether, for some functions, there is a provable gap between the time needed to stably compute the functions with an output-oblivious CRN and the time needed by a CRN that is not restricted to be output-oblivious. Yet other directions are to better understand output-oblivious CRN computation when errors are allowed, and whether it is possible to "repair" a CRN that is not output-oblivious so that composition is possible.

References

1 Dan Alistarh, Rati Gelashvili, and Milan Vojnović. Fast and exact majority in population protocols. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 47–56. ACM, 2015.

2 Dana Angluin, James Aspnes, and David Eisenstat. Fast computation by population protocols with a leader. In *Dolev S. (eds) Distributed Computing (DISC), Lecture Notes in Computer Science*, volume 4167, pages 61–75. Springer, Berlin, Heidelberg, 2006.

3 Dana Angluin, James Aspnes, and David Eisenstat. Stably computable predicates are semilinear. In *PODC '06: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 292–299, New York, NY, USA, 2006. ACM Press. doi:10.1145/1146381.1146425.

4 Dana Angluin, James Aspnes, David Eisenstat, and Eric Ruppert. The computational power of population protocols. *Distributed Computing*, 20(4):279–304, 2007.

5 Adam Arkin, John Ross, and Harley H McAdams. Stochastic kinetic analysis of developmental pathway bifurcation in phage λ-infected Escherichia coli cells. *Genetics*, 149(4):1633–1648, 1998.

6 Luca Cardelli and Attila Csikász-Nagy. The cell cycle switch computes approximate majority. *Scientific reports*, 2:656, 2012.

7 Cameron Chalk, Niels Kornerup, Wyatt Reeves, and David Soloveichik. Composable Rate-Independent Computation in Continuous Chemical Reaction Networks. In Milan Ceska and David Safránek, editors, *Computational Methods in Systems Biology*, pages 256–273, Cham, 2018. Springer International Publishing.

8 Ho-Lin Chen, David Doty, and David Soloveichik. Deterministic function computation with chemical reaction networks. *Natural Computing*, 13(4):517–534, December 2014.

9 Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, and Eric Ruppert. Secretive birds: Privacy in population protocols. In *International Conference On Principles Of Distributed Systems*, pages 329–342. Springer, 2007.

10 Leonard Eugene Dickson. Finiteness of the Odd Perfect and Primitive Abundant Numbers with n Distinct Prime Factors. *American Journal of Mathematics*, 35(4):413–422, 1913. URL: http://www.jstor.org/stable/2370405.

11 David Doty and Monir Hajiaghayi. Leaderless deterministic chemical reaction networks. *Natural Computing*, 14(2):213–223, 2015.

12 Allen Hjelmfelt, Edward D Weinberger, and John Ross. Chemical implementation of neural networks and Turing machines. *Proceedings of the National Academy of Sciences*, 88(24):10983–10987, 1991.

13 Hua Jiang, Marc D Riedel, and Keshab K Parhi. Digital signal processing with molecular reactions. *IEEE Design and Test of Computers*, 29(3):21–31, 2012.

14 Nils E Napp and Ryan P Adams. Message passing inference with chemical reaction networks. In *Advances in neural information processing systems*, pages 2247–2255, 2013.

15 Lulu Qian and Erik Winfree. Scaling up digital circuit computation with DNA strand displacement cascades. *Science*, 332(6034):1196–1201, 2011.

16 David Soloveichik, Matthew Cook, Erik Winfree, and Jehoshua Bruck. Computation with finite stochastic chemical reaction networks. *Natural Computing*, 7, 2008.

17 David Soloveichik, Georg Seelig, and Erik Winfree. DNA as a universal substrate for chemical kinetics. *Proceedings of the National Academy of Sciences*, 107(12):5393–5398, 2010.

18 Chris Thachuk and Anne Condon. Space and energy efficient computation with DNA strand displacement systems. In *International Workshop on DNA-Based Computers*, pages 135–149. Springer, 2012.

19 David Yu Zhang, Rizal F Hariadi, Harry MT Choi, and Erik Winfree. Integrating DNA strand-displacement circuitry with DNA tile self-assembly. *Nature communications*, 4:1965, 2013.

The Synergy of Finite State Machines

Yehuda Afek[1]
Tel Aviv University, Tel Aviv, Israel
afek@cs.tau.ac.il

Yuval Emek[2]
Technion - Israel Institute of Technology, Haifa, Israel
yemek@technion.ac.il

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

Abstract

What can be computed by a network of n randomized finite state machines communicating under the *stone age* model (Emek & Wattenhofer, PODC 2013)? The inherent linear upper bound on the total space of the network implies that its global computational power is not larger than that of a randomized linear space Turing machine, but is this tight? We answer this question affirmatively for bounded degree networks by introducing a stone age algorithm (operating under the most restrictive form of the model) that given a designated *I/O node*, constructs a *tour* in the network that enables the simulation of the Turing machine's tape. To construct the tour with high probability, we first show how to 2-*hop color* the network concurrently with building a spanning tree.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases finite state machines, stone-age model, beeping communication scheme, distributed network computability

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.22

Related Version A full version [6] that contains all missing proofs along with some additional material can be obtained from `http://yemek.net.technion.ac.il/files/tsfsm.pdf`.

1 Introduction

Synergy, the whole is greater than its parts, is many times true, however in traditional distributed computing, each node is usually assumed to be as powerful as a Turing machine, hence its local computational power is equivalent to the global computational power of the whole network. Here, we address the computational power of a network of *randomized finite state machines* with a very weak communication scheme (similar to the communication scheme of the *beeping* model, [14, 2]), and show that even under these harsh conditions, synergy can be achieved: *the network as a whole is computationally more powerful than its individual nodes.*

Recently, there is a growing interest in the study of networks of sub-silicon devices, including biological networks [1, 29, 36] and networks of man-made nano-devices [35, 19, 11],

[1] The work of Y. Afek was partially supported by a grant from the Blavatnik Cyber Security Council and the Blavatnik Computer Science Research Fund.
[2] The work of Y. Emek was supported in part by an Israeli Science Foundation grant number 1016/17.

© Yehuda Afek, Yuval Emek, and Noa Kolikant;
licensed under Creative Commons License CC-BY
22nd International Conference on Principles of Distributed Systems (OPODIS 2018).
Editors: Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira; Article No. 22; pp. 22:1–22:16

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

through the lens of theoretical distributed computing. These are typically huge networks of primitive devices that nevertheless perform complicated tasks (e.g., an ant colony that solves problems no small number of ants can), thus raising the following question: How do limitations on the local computation and communication capabilities of the individual nodes affect the global computational power of the whole network?

The current paper addresses this question using the *stone age (SA)* model of Emek and Wattenhofer [25] that captures a network of devices with very weak local computation and communication capabilities. It has been shown in [25, Sec. 5 (full version)] that an n-node SA network with a path topology can simulate a randomized $O(n)$-space Turing machine, denoted hereafter as an RSPACE(n) machine. Little is known though about the global computational power of SA networks with more general topologies and/or more restrictive communication schemes. In this paper, we shed some light into this unexplored research domain, proving that RSPACE(n) machines can be simulated over any network topology of bounded degree by a variant of the SA model where the nodes have no sender collision detection (see Section 1.1).

1.1 Model

Our model follows the *stone age (SA)* model introduced in [25] and used subsequently in [26, 5]. Throughout, we assume that the communication network is represented by a finite size connected undirected graph $G = (V, E)$ with node degrees bounded by constant Δ. The nodes are controlled by *randomized finite automata* with state space Q, message alphabet Σ, and transition function τ whose role is explained soon.

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

The execution is event driven with an asynchronous scheduler that schedules the aforementioned message delivery events as well as node activation events.[3] When node $v \in V$ is activated, the transition function $\tau : Q \times \{0,1\}^{\Sigma} \to 2^{Q \times \Sigma}$ determines (in a probabilistic fashion) its next state $q' \in Q$ and the next message $\sigma' \in \Sigma$ to be sent based on its current state $q \in Q$ and the current content of its ports.[4] Formally, the pair (q', σ') is chosen uniformly at random from $\tau(q, \chi_v)$, where $\chi_v \in \{0,1\}^{\Sigma}$ is defined so that $\chi_v(\sigma) = 1$ if and only if σ is written in at least one port $\psi_v(\cdot)$.

To complete the definition of the randomized finite automata, one has to specify the initial state $q_0 \in Q$ (assumed to be the same for all nodes in the current paper), the set $Q_{out} \subseteq Q$ of output states that encode the node's output, and the initial message $\sigma_0 \in \Sigma$ written in the ports when the execution begins. In addition, SA algorithms are required to have *termination detection*, namely, every node must eventually decide on its output and this decision is irrevocable.

[3] The only assumption we make on the event scheduling is FIFO message delivery: a message sent by node u at time t is written into port $\psi_v(u)$ of its neighbor v before the message sent by u at time $t' > t$.
[4] Node v's actual transition in step t is an atomic operation occurring at the beginning of the step.

Following the convention in message passing distributed computing (cf. [37]), the *run-time* of an asynchronous SA algorithm is measured in terms of *time units* scaled to the maximum time it takes to deliver any message or the time between any two consecutive activations of a node. Refer to [25] for a more detailed description of the SA model.

We adopt the communication scheme presented in [5] that weakens the SA model of [25] in two aspects. First, under the model of [25], the algorithm designer could choose an additional constant *bounding parameter* $b \in \mathbb{Z}_{>0}$, providing the nodes with the capability to count the number of ports holding message $\sigma \in \Sigma$ up to b. As done in [5], in the current paper, the bounding parameter is set to $b = 1$. This model choice can be viewed as an asynchronous multi-frequency variant of the *beeping* communication model [14, 2].

Second, in contrast to the setting considered in [25], the communication graph $G = (V, E)$ assumed in the current paper may include *self-loops* of the form $(v, v) \in E$ which means, in accordance with the aforementioned model definition, that node v admits port $\psi_v(v)$ that holds the last message received from itself. Using the terminology of the beeping model literature (see, e.g., [2]), the assumption that the communication graph is free of self-loops corresponds to a *sender collision detection*, whereas lifting this assumption means that node v may not necessarily distinguish its own transmitted message from those of its neighbors $u \neq v$.

The cruxes of the SA model are that (1) node v cannot distinguish between its ports; and (2) the number of states in Q and the size of the message alphabet Σ are constants independent of the size (and any other global parameter) of the graph G.[5] The same holds for the size of the transition function τ, that can be encoded as a $Q \times 2^{\Sigma}$ table whose entries are subsets of $Q \times \Sigma$.

Sequential Stone Age Machines. We wish to use a SA network to simulate an RSPACE(n) machine $\mathcal{M}$, but before we can describe this simulation, we have to explain how the $O(n)$-bit input I of $\mathcal{M}$, that is normally stored in $\mathcal{M}$'s tape at the beginning of the execution, is provided to our network. Clearly, no node in the network can hold more than a constant number of bits, hence I should be distributed over multiple nodes. Unlike [25], where a path topology is assumed, here the network topology is arbitrary and does not (initially) induce any sequential order on the nodes, thus storing I in the nodes of the network before the execution starts does not make sense. Instead, we introduce the key notion of a *sequential stone age machine (SSAM)*, where I is fed to the SA algorithm bit-by-bit in a sequential fashion through some node.

Formally, given a network $G = (V, E)$, a SSAM is a SA algorithm running on the nodes of G that allows an external user to

(1) pick any node $v \in V$ and send to it a designated `I/O_prepare` message;
(2) wait until v sends a designated `I/O_ready` message;
(3) feed v with a sequence of input bits by means of sending a sequence of designated input messages (and receiving a corresponding sequence of acknowledgments from v);
(4) wait until the computational process terminates; and
(5) get the desired output back from v by means of receiving from it a sequence of designated output messages.

We refer to node v picked by the user in (1) as an *I/O node*. To exploit the combined computational power of all nodes (in contrast to a single node whose computational power is restricted to that of a randomized finite state machine), the computational process described

[5] In the current paper, $|Q|$ and $|\Sigma|$ may depend on the fixed degree bound Δ; the exact dependency is analyzed in the full version [6].

in (4) typically involves the whole network. The SSAM is said to be a (T^p, T^{io})-*SSAM* if it is guaranteed that the external user waits at most T^p time between sending the `I/O_prepare` message and receiving the `I/O_ready` message and at most T^{io} time between feeding the input bits and getting back the output bits.

1.2 Our Results

We prove that any problem that can be solved whp by an RSPACE(n) machine in time T can be solved whp on any n-node bounded degree graph G by an $(O(D), O(T))$-SSAM operating under an asynchronous scheduler, where D denotes the diameter of G;[6] in other words, it takes $O(D)$ time to initialize the SSAM so that it is ready to accept its input, whereas the actual simulation of the RSPACE(n) machine takes $O(T)$ time. Specifically, our main algorithmic contribution is a SA algorithm that given an n-node bounded degree graph $G = (V, E)$ and a designated *root* node $r \in V$, constructs a 2-*hop coloring* of G and a node sequence $\langle S(i) \rangle_{i=0}^{2n-1}$, referred to as a *tour*, that satisfies: (i) every node appears in S exactly twice; (ii) $S(0) = S(2n-1) = r$; and (iii) the state of node $S(i)$ encodes enough information to route a message to $S(i+1 \mod 2n)$ and to $S(i-1 \mod 2n)$ that reaches its destination in $O(1)$ time for every $0 \leq i \leq 2n-1$; our algorithm terminates with a correct 2-hop coloring and a correct tour in time $O(D)$ whp.

In the SSAM context, the tour S is constructed during phase (2) (while the external user waits for the `I/O_ready` message) with the I/O node serving as the root. This tour is then employed to simulate a randomized Turing machine $\mathcal{M}$ with a $(2n)$-cell tape in phase (4) as follows. Node $S(i)$ simulates the ith cell $C(i)$ in the tape so that the state of $S(i)$ encodes: (a) the current content of $C(i)$; (b) whether the head of $\mathcal{M}$ is currently located at $C(i)$; and (c) the state of $\mathcal{M}$ in case the head is located at $C(i)$. A step of $\mathcal{M}$'s (sequential) execution, where the head moves from cell $C(i)$ to cell $C(i \pm 1)$, is implemented by sending a designated 'head moving' message from $S(i)$ to $S(i \pm 1)$ that encodes the new state of $\mathcal{M}$.

1.3 Main Technical Challenges

A 2-hop coloring is a useful construction in anonymous networks (see, e.g., [27]) that enables local point-to-point communication under broadcast communication schemes. As discussed in [25, Sec. 4.3], it is fairly easy to design a 2-hop coloring SA algorithm in bounded degree graphs with bounding parameter $b = 2$. However, once the bounding parameter is set to $b = 1$ (as defined in the current paper), this becomes a challenging task because the nodes can no longer verify (deterministically) that their neighborhood does not admit color conflicts. The setting considered in the current paper is even harder since the graph may contain self-loops.

Our algorithm resolves this issue by coloring the nodes concurrently with growing a tree $\widetilde{T}$ of depth $O(D)$ rooted at the designated root r. The nodes use a randomized test that looks for color conflicts and if a conflict is detected, the tree $\widetilde{T}$ is carefully used to reset the coloring and tree construction processes. It is interesting to point out that without a designated root, it is impossible to obtain even a 1-hop coloring in our setting – see Section 5.

Another source of difficulty that we had to overcome when designing our algorithm stems from the requirement that the algorithm terminates correctly whp. While whp guarantees are common in traditional distributed graph algorithms, they are more challenging to obtain with SA algorithms: the individual nodes do not (and cannot) have any notion of n; nevertheless, the algorithm should err with probability that decreases (polynomially) with n.

---

[6] Throughout this paper, we say that event A occurs *with high probability*, abbreviated by *whp*, if $\mathbb{P}(A) \geq 1 - n^{-c}$, where n is the number of nodes in the graph and c is an arbitrarily large constant.

1.4 Related Work

Most literature on distributed network algorithms does not deal with computability issues, simply because the computational power of the whole network is identical to that of a single node (that is, a Turing machine). Things become more interesting when restrictions are imposed on the computational power of the individual nodes, in particular, when the nodes are restricted to finite state machines.

Computational models based on networks of finite state machines have been studied for many years. The best known such model is the extensively studied *cellular automata* that were introduced by Ulam and von Neumann [39] and became popular with Martin Gardner's Scientific American column on Conway's *game of life* [31] (see also [40]).

Another popular model that considers a network of finite state machines is the *population protocols* model, introduced by Angluin et al. [7] (see also [8, 35]), where the network entities communicate through a sequence of atomic pairwise interactions controlled by a fair (adversarial or randomized) scheduler. This model provides an elegant abstraction for networks of mobile devices with proximity derived interactions and it also fits certain types of chemical reaction networks [21].

The neat *amoebot model* introduced by Dolev et al. [20] also considers a network of finite state machines in a (hexagonal) grid topology, but in contrast to the models discussed so far, the particles in this network are augmented with certain mobility capabilities, inspired by the amoeba contraction-expansion movement mechanism. This model has been successfully employed for the theoretical investigation of self-organizing particle systems [38, 18, 16, 19, 17, 11, 15], especially in the context of *programmable matter*.

The SA model, that constitutes the heart of this paper, was introduced in [25] with the goal of demonstrating that certain classic distributed graph problems, e.g., maximal independent set, coloring, and maximal matching, can be solved fast (in polylogarithmic time) by a network of devices whose computation and communication capabilities are perhaps sufficiently weak to capture biological cellular networks (cf. [10]). Since then, this model was successfully employed in theoretical studies of distributed computing in other kinds of networks including a series of works inspired by ant foraging processes [34, 23, 33, 22, 12] as modeled by Feinerman et al. [30, 28]. The SA maximal independent set algorithm of [25] has been adjusted to handle dynamic networks too [24].

As discussed in Section 1.2, the SA algorithm developed in the current paper assumes a unique root node, where in the SSAM context, the role of the root node is played by the I/O node chosen by the external user. The task of selecting a unique root node (cf. *leader election*) under the SA model has recently been studied in [5]. This paper also introduces the variant of the SA model (with a weaker communication scheme, see Section 1.1) used in the current paper.

Much of the technical challenge in designing the SA algorithms presented in [25, 24, 5] stems from the necessity to handle graphs of arbitrary node degrees. In contrast, the focus of the current paper (that studies a completely different problem) is restricted to bounded degree graphs and the main technical challenge arises from other factors (see Sec. 1.3). In this regard, it is important to point out that although imposing a constant bound on the node degrees is a severe limitation from a graph theoretic perspective, in practice, many of the networks that motivate our model, particularly biological cellular networks, typically exhibit small node degrees.

2 Preliminaries

Notation and Terminology. Throughout this paper, the degree bound Δ is regarded as a constant. In some places, the asymptotic notation is augmented with a Δ subscript to emphasize that a Δ function is hidden, e.g., $O_{\Delta}(n)$ or $\Omega_{\Delta}(\log n)$. Using this notation, notice that the diameter D of the graph is larger than $\log_{\Delta} n = \Omega_{\Delta}(\log n)$. When the exact dependency on Δ is more important, we may also mention it explicitly inside the asymptotic notation.

We denote the shortest distance (in hops) from node u to node v in graph $G = (V, E)$ by $\text{dist}_G(u, v)$ and omit the subscript G when the graph is clear from the context. The *(inclusive) d-hop neighborhood* of v is denoted by $\Gamma^{d+}(v) = \{u \in V \mid \text{dist}(v, u) \leq d\}$; when $d = 1$, we omit it from the superscript and write simply $\Gamma^{+}(v)$. Let $\Gamma(v) = \Gamma^{+}(v) - \{v\}$ be the *(exclusive) neighborhood* of v. Let $\Gamma^{*}(v) = \Gamma^{+}(v)$ if v admits a self-loop and $\Gamma^{*}(v) = \Gamma(v)$ otherwise. A *k-hop coloring* of G is a function $c : V \to \mathbb{Z}_{>0}$ satisfying the requirement that $c(u) \neq c(v)$ for every two nodes $u \neq v$ such that $\text{dist}_G(u, v) \leq k$. A 1-hop coloring is often referred to simply as a *coloring*.

In the context of rooted trees, the terms parent, child, sibling, leaf, depth, and height are used in their standard meaning (see, e.g., [13]). We also use the term *branch* for a tree path that starts at the root. These terms are generalized from a rooted tree to a *directed acyclic graph (DAG)*, recalling that there, the parent of node v and the branch that ends at v are not necessarily unique. Finally, for a positive integer k, the set $\{1, 2, \ldots, k\}$ is denoted by $[k]$.

Synchronizer Transformation. As explained in Section 1.1, the execution is controlled by an asynchronous scheduler. One of the contributions of [25] is a SA *synchronizer* implementation (cf. the α-synchronizer of Awerbuch [9]). Given a synchronous SA algorithm $\mathcal{A}$ whose execution progresses in fully synchronized *rounds* $t \in \mathbb{Z}_{>0}$ (with simultaneous wake-up), the synchronizer generates a valid (asynchronous) SA algorithm $\mathcal{A}'$ whose execution progresses in *pulses* such that the actions taken by $\mathcal{A}'$ in pulse t are identical to those taken by $\mathcal{A}$ in round t.[7] The synchronizer is designed so that the asynchronous algorithm $\mathcal{A}'$ has the same bounding parameter b ($= 1$ in the current paper) and asymptotic run-time as the synchronous algorithm $\mathcal{A}$.

Although the model considered by Emek and Wattenhofer [25] assumes that the graph has no self-loops, it is straightforward to apply their synchronizer to graphs that do include self-loops (see also [5]). Hence, our algorithm is designed to operate under a locally synchronous scheduler. Specifically, we assume that the execution progresses in synchronous rounds $t \in \mathbb{Z}_{>0}$, where in round t, each node v

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

Notice that under a locally synchronous scheduler, if node u is in round t and node v is in round t', then it is guaranteed that $|t - t'| \leq \text{dist}(u, v)$. This means that there exists some constant c so that in any period of $c(T + D)$ time units, every node in the graph performs at least T steps. Consequently, an upper bound of $O(T)$ on the number of rounds performed by an arbitrary node, implies an upper bound of $O(T + D)$ on the total elapsed time.

[7] We emphasize the role of the assumption that when the execution begins, the ports hold the designated initial message σ_0. Based on this assumption, a node can "sense" that some of its neighbors have not been activated yet, hence synchronization can be maintained right from the beginning.

3 Algorithm Description

In this section, we present our algorithm that constructs the desired $(2n)$-hop long tour over the bounded degree graph assuming there is a single distinguished root node.

Overview. At the beginning, all nodes (other than the root) are identical, and as a first step, the algorithm 2-hop colors them. Given a 2-hop coloring, each node can distinguish between its neighbors and establish a one-to-one communication with each of them. Based on this infrastructure, it is possible to construct a rooted tree from the distinguished root and build a tour on it.

To build the above on a network of identical randomized finite state machines, we use a *layered approach*. The first layer, referred to as the *synchronizer* layer, runs a synchronizer similar to those designed in [3, 25, 5] so that all layers above it operate assuming a locally synchronous scheduler. The goal of the second layer, referred to as the *degree estimation* layer, is for each node to compute its own degree in G. This is done based on a randomized process that continuously verifies that the current estimate on the number of neighbors is correct. Each time the estimate is updated, a *reset* is invoked and the layers above this second layer may be restarted (more on that later).

In the next, third, 2-*hop coloring* layer, we rely on each node correctly knowing its degree. The root starts a process that 2-hop colors the graph and constructs a rooted spanning tree, denoted by $\widetilde{T}$, at the same time. Upon termination of this process, an echo process up the branches of $\widetilde{T}$ (towards the root) is invoked. Tree $\widetilde{T}$ is designed so that its depth is $O(D)$ whp. Notice that $\widetilde{T}$'s construction assumes all nodes know their accurate degree; if not, the constructed $\widetilde{T}$ may be a DAG (see Section 2), but this will be detected by the degree estimation layer and a reset will (eventually) be invoked to delete $\widetilde{T}$ and restart the tree construction (intertwined with 2-hop coloring) process. The goal of the fourth and final layer, referred to as the *tour construction* layer, is to construct the desired tour. This is done by simulating a *depth first search* traversal of $\widetilde{T}$ in a concurrent manner. The depth first search procedure is implemented so that when the root terminates, the tour is ready to be used. Refer to the full version [6] for an illustration of the layer hierarchy.

3.1 The Layers

Recall that initially, all nodes reside in the initial state q_0 and all ports hold the initial letter σ_0. The root starts the synchronizer (as described in Section 2) and any node that reads a message different than σ_0 on any of its ports, starts executing the synchronizer layer and the layers above it as described below. The bottom layer, the synchronizer, is the only asynchronous layer. All layers above it operate under a locally synchronous scheduler and make a transition on a round-by-round basis.

The layers model essentially breaks the state machine in each node into several randomized finite state machines, each corresponding to a *block* of the algorithm, where block in this regard refers to a layer or a process within a layer. The state set Q and message alphabet Σ are the Cartesian products of the state sets and message alphabets, respectively, of each block's state machine. Some blocks make no move until another block enters some specified state indicating that its task has been completed (for example, all nodes in the neighborhood of node v must be colored before v can start participating in the tour construction layer).

Below we provide detailed descriptions of the different blocks (refer to the full version [6] for pseudocodes). For clarity of the exposition, these blocks are described in a conventional

(distributed) algorithmic style, but they can be easily implemented using a randomized finite state machine. In the full version [6], we discuss the size of the state sets and message alphabets of those state machines in more detail.

3.1.1 The Degree Estimation Layer

In each round (working on top of the synchronizer), each node v first randomly draws a label l_v (the word "color" is preserved for the 2-hop coloring procedure) from a set of Δ^4 labels and then, verifies that the number of distinct labels chosen by its neighbors is not larger than its current `deg_estimate` variable. Otherwise, it updates `deg_estimate` and interrupts the layer above it. These labels are also used in upper layers but are only useful when then number of current labels matches the most updated `deg_estimate`. For this reason we set another flag variable `safe` which is used to indicate if the current number of labels equals `deg_estimate`.

3.1.2 The 2-Hop Coloring Layer

A naive algorithm would be for each node in each round to randomly select a color from a sufficiently large (as a function of Δ) palate of colors until it verifies that the number of colors in its neighborhood is not smaller than the degree estimate given by the layer below. However, neighbors may have to change their color causing a chain of node re-coloring that may be difficult to control. Instead, we carefully use the network's size to our benefit, producing a 2-hop coloring whp. To that end, we *intertwine* the coloring trials with a tree construction process, where each node is colored before joining the tree; this tree, denoted by $\widetilde{T}$, serves as the underlying communication structure if a reset process is invoked (this will be explained later). Using a BFS like tree construction, we guarantee that the depth of $\widetilde{T}$, as well as its construction time, are $O(D)$ whp. However, the (randomized) 2-hop coloring trials, carried out by the nodes on the boundary of $\widetilde{T}$, may introduce some level of asynchronicity and thus, $\widetilde{T}$ may not be an exact BFS.

The Tree Construction Process. The process of constructing $\widetilde{T}$ while 2-hop coloring the nodes on its boundary is referred to as the *tree construction* process. The main variables maintained by this process for each node v are `color` and `p_color`, storing the color of v and the color of v's parent(s) in $\widetilde{T}$, respectively (the plural form corresponds to the case where $\widetilde{T}$ is a DAG, rather than a tree). The values of these variables are reported in every message sent by v.

The tree construction process starts with the root r setting $r.\texttt{color} \leftarrow 1$; following that, the root transmits a designated `join` message. Node v receiving a `join` message for the first time, initiates its 2-hop coloring trials; when these trials succeed, v joins $\widetilde{T}$.

Specifically, upon receiving the `join` message for the first time, node v selects the color of one of the neighbors $u \in \Gamma(v)$ from which a `join` message was received; it then writes $v.\texttt{p_color} \leftarrow u.\texttt{color}$ and $v.\texttt{g_p_color} \leftarrow u.\texttt{p_color}$, where `g_p_color` is another (temporary) variable maintained by v. Following that, v randomly selects a new color c from the palette $[\Delta^4 - \{v.\texttt{p_color}, v.\texttt{g_p_color}\}]$ and broadcasts a request to color itself with color c using a designated $\texttt{color_req}(c)$ message. Node v then waits (for two rounds) for its neighbor's responses (the process in charge of these responses is described soon), and if receives `approve` messages from all of them, then it writes $v.\texttt{color} \leftarrow c$ and broadcasts a `join` message; otherwise (some neighbors of v did not respond with an `approve` message), it starts a new coloring trial. Once v's `color` variable is set, we think of it as being part of $\widetilde{T}$. Notice that the nodes do not record their children in $\widetilde{T}$.

The Color Approval Process. The mechanism in charge of approving/disapproving the `color_req` messages is referred to as the *color approval* process. This process runs continually at each node v and its role is to check that when node $u \in \Gamma(v)$ attempts to pick the color c (reflected by a $\texttt{color_req}(c)$ message received from u), no other node in $\Gamma^+(v)$ is (or soon to be) colored c. This task is trivial to accomplish if all the nodes in $\Gamma^+(v) - \{u\}$ are already colored. However, it may be the case that multiple neighbors of v may have not yet chosen a color (which means that they have not yet joined $\widetilde{T}$) and thus, may look indistinguishable to v. What if two (or more) of these neighbors attempt to pick the same color c?

To overcome this problem, we go back to the `safe` flag of the degree estimation layer. Recall that this flag is set to true only when the number of distinct labels received by node v is equal to its `deg_estimate` variable, which means that v can distinguish between all its perceived neighbors. Therefore, v sends an `approve` message if (and only if) $\texttt{safe} == \texttt{true}$ and the colors it receives in the `color_req` messages are consistent with a valid 2-hop coloring given the fixed colors in $\Gamma^+(v)$; that is, the `color_req` colors are distinct and different from the colors of the nodes in $\Gamma^+(v)$ that already fixed their colors.

The Echo Process. An *echo* process up the branches of $\widetilde{T}$ is used to detect the termination of the tree construction process. Each node v can detect the colors of its children in $\widetilde{T}$ – those are simply the nodes declaring $v.\texttt{color}$ as their parent's color. When a designated `echo` message is received from all its children (at the same round), v starts sending `echo` messages until its parent sends its own `echo` message. The execution of the echo process together with the tree construction process forces all the nodes of the network to execute at least D rounds, which is critical to guarantee the high probability success of the degree estimation layer as $D > \log_{\Delta} n$.

3.1.3 The Tour Construction Layer

The tour construction layer lays out a $(2n)$-hop long tour of the network based on the $2n$ *timestamps* of a *depth first search (DFS)* traversal of $\widetilde{T}$ (see, e.g., [13, Section 22.3]). Recall that these timestamps are integers in $\{0, 1, \ldots 2n-1\}$ and that the DFS traversal assigns two of them to each node v: one for the time $0 \leq v.d \leq 2n-2$ at which v is discovered by the DFS traversal and one for the time $0 \leq v.f \leq 2n-1$ at which the DFS traversal backtracks from v. We construct the desired tour $\langle S(i) \rangle_{i=0}^{2n-1}$ so that $S(v.d) = S(v.f) = v$.

The DFS timestamps have the following important property: if one of the timestamps of node u is in $\{v.d \pm 1, v.f \pm 1\}$, then u is either v itself, the parent of v, a child of v, or a sibling of v.[8] The nodes cannot store the actual timestamps (this would have required $\Omega(\log n)$ bits), however, node $S(i)$ can store the colors of the nodes along the unique paths in $\widetilde{T}$ from $S(i)$ to $S(i \pm 1)$. If the 2-hop coloring is valid, then this information enables one-to-one communication between $S(i)$ and $S(i \pm 1)$. The aforementioned property ensures that each one of these two paths includes at most two hops, so in total, the tour construction layer stores at most 8 colors of the palette $[\Delta^4]$ at each node v.

These colors are stored in the designated $\texttt{forward_pointer}^d$, $\texttt{backward_pointer}^d$, $\texttt{forward_pointer}^f$, and $\texttt{backward_pointer}^f$ variables that correspond to $S(v.d+1)$, $S(v.d-1)$, $S(v.f+1)$, and $S(v.f-1)$, respectively. Each pointer encodes the empty path from v to itself or the path to a child, parent, or sibling of v and thus, consists of at most two colors.

[8] The arithmetic done in this section is modulo $2n$.

The role of the tour construction layer is to set those pointers so that the resulting tour reflects the DFS traversal of $\widetilde{T}$. This is done concurrently at all nodes, without actually running a DFS traversal. This construction could have been totally local, without sending any message, if the nodes would have been fully aware of the color assignment in their 2-hop neighborhood in $\widetilde{T}$. However, this would have been expensive in terms of the size $|Q|$ of the state set (making it exponential, rather than polynomial, in Δ – see the full version [6]). Instead, the 2-hop coloring layer constructs $\widetilde{T}$ so that each node stores only its parent. To overcome this obstacle, each node v instructs its children on how to set their pointers, helping them connect to their siblings and to v itself, without knowing all of their colors at the same time. The method employed in this regard is iterative, trading space for time, by connecting one child node at a time. This method is invoked at v right after the echo process has ended (and v stopped sending `echo` messages).

3.1.4 Resets

The algorithm described thus far may err if a node in $\widetilde{T}$ under-estimated its degree. In this case the 2-hop coloring may be invalid which means that $\widetilde{T}$ may be a DAG (rather than a tree) and the tour constructed based on $\widetilde{T}$ may be wrong. Therefore, if a degree under-estimation is detected, the data structures of the 2-hop coloring and tour construction layers must be deleted and re-constructed from scratch. This can be detected in two possible ways: (1) when a node in the degree estimation layer increases its `deg_estimate` variable; or (2) when a node receives different messages with the same `color` field (which means that they come from different neighbors that were not distinguished so far). The former condition is checked by the degree estimation layer. In both cases, if a degree under-estimation is detected, a designated `reset` interrupt is signaled.

The process that catches the `reset` interrupts is referred to as the *reset* process. As mentioned earlier, this process resets the flags and variables maintained by the 2-hop coloring and tour construction layers. (We emphasize that the `deg_estimate` variable and the variables maintained by the synchronizer layer are not affected by a reset.) But before these flags and variables can be reset at node v, we must make sure that the the reset process is invoked at all other nodes in $\widetilde{T}$. This can be tricky as an independent `reset` interrupt may be signaled at some node u while the reset process initiated by v spreads in the network.

To overcome this obstacle, we follow the reset technique introduced in [4], and execute the reset process as follows. (1) A node receiving a `reset` interrupt initiates a `reset_request` only if it has already joined $\widetilde{T}$ (otherwise no reset is initiated by this node). (2) The `reset_request` messages are forwarded up the branches of $\widetilde{T}$ (towards the root). (3) Upon receiving a `reset_request` message, the root broadcasts `freeze_command` messages that are forwarded to all nodes in $\widetilde{T}$ down its branches. (4) Upon receiving a `freeze_command` message, each leaf echos a `freeze_ack` message; each internal node sends a `freeze_ack` message to its parent after it receives `freeze_ack` messages from all its children. (5) When the root is fully acknowledged with the `freeze_ack` messages (and thus knows the entire $\widetilde{T}$ is frozen), it broadcasts `reset_command` messages that are forwarded to all nodes in $\widetilde{T}$ down its branches. (6) Upon receiving a `reset_command` message, each leaf echos a `reset_ack` message; each internal node sends a `reset_ack` message to its parent after it receives `reset_ack` messages from all its children. At this stage, the (leaf or internal) node resets all flags and variables of the 2-hop coloring and tour construction layers. When the root is fully acknowledged with the `reset_ack` messages, it restarts the 2-hop coloring and tour construction layers from scratch.

Notice that the nodes reset and remove themselves from $\widetilde{T}$ from the leaves up towards the root, so $\widetilde{T}$ never decomposes into several subgraphs. Moreover, the freeze phase in the reset process ensures that $\widetilde{T}$ does not continue growing in an uncontrolled manner while a reset is in motion. Finally, notice that the reset process works as long as $\widetilde{T}$ is a DAG and does not rely on a tree topology.

4 Analysis

Due to space limitations, this extended abstract contains only parts of the analysis; refer to the full version [6] for the complete analysis. Recall that the 2-hop coloring layer generates a subgraph $\widetilde{T}$ that consists of all colored nodes with an edge connecting vertex u to vertex v if $u.\texttt{color} = v.\texttt{p_color}$; for the sake of convenience, we orient this edge from u to v and think of $\widetilde{T}$ as a directed graph.

▶ **Lemma 1.** *$\widetilde{T}$ is a DAG. Moreover, r is the unique source in $\widetilde{T}$.*

4.1 Correctness with High Probability

The desired tour construction relies on the correctness of the 2-hop coloring. The latter allows node v to communicate in a one-to-one fashion with each of its neighbors u, which in turn, supports one-to-one communication with more distant nodes w provided that the colors along some (v, w)-path are known. The correctness (whp) of our 2-hop coloring scheme is based on three foundations:

(1) Once all nodes hold an accurate degree estimation, a valid 2-hop coloring is reached with no further resets with probability 1 (see Corollary 4).

(2) After $O(\log_\Delta(n))$ rounds of the degree estimation layer, each node holds an accurate degree estimation whp (see Corollary 6).

(3) All nodes execute $\Omega(\log_\Delta(n))$ rounds of the degree estimation layer before termination (see Lemma 7).

The following lemma plays a key role in establishing these three foundations.

▶ **Lemma 2.** *Consider a tree construction process that starts after all nodes hold an accurate degree estimation. Let w be some node and let $u, v \in \Gamma^+(w)$, $u \neq v$, be two nodes in $\widetilde{T}$. Then, $u.\texttt{color} \neq v.\texttt{color}$.*

Proof. Nodes u and v are colored since both are in $\widetilde{T}$. Assume by contradiction that $u.\texttt{color} = v.\texttt{color} = c$. Recall that a node can fix its own color only in a round at which it receives `approve` messages from all its neighbors. Let t_u and t_v be these rounds for u and v, respectively, from the perspective of w (that is, in w's round counting). Assume without loss of generality that $t_u \leq t_v$.

If $t_u = t_v$, then w approved $u.\texttt{color} \leftarrow c$ and $v.\texttt{color} \leftarrow c$ in the same round. The design of the color approval process guarantees that $w.\texttt{safe}$ was `true` at that round, which means that the number of distinct labels received by w at that round was $w.\texttt{deg_estimate}$. The assumption that all `deg_estimate` variables were accurate when the tree construction process started implies that u and v picked distinct labels. But this means that the color approval process at w witnessed a color conflict in that round and did not send an `approve` message, reaching a contradiction.

If $t_u < t_v$, then w approved $v.\texttt{color} \leftarrow c$ after u has already fixed its color to $u.\texttt{color} = c$. But the design of the color approval process guarantees that if v tries to pick color c, then w does not send an `approve` message, again reaching a contradiction. ◀

▶ **Corollary 3.** *If a tree construction process starts after all nodes hold an accurate degree estimation, then $\widetilde{T}$ will not experience a reset.*

Proof. A reset is initiated either because some node updates its `deg_estimate` variable (the degree estimation layer), which cannot happen due to the assumption, or because two different messages carrying the same `color` field are received (the validity check process), which cannot happen by Lemma 2. ◀

▶ **Corollary 4.** *Once all nodes hold an accurate degree estimation, a valid* 2*-hop coloring is reached within finite time with probability* 1*.*

▶ **Observation 5.** *For a label set of size Δ^4, in every round of the degree estimation process, the number of distinct labels received by node v equals its degree with probability $1 - O(1/\Delta^2)$.*

Proof. Follows immediately from a birthday paradox argument since $|\Gamma^*(v)| \leq \Delta + 1$. ◀

▶ **Corollary 6.** *For a label set of size Δ^4, after $O(\log_\Delta(n))$ rounds of node v's degree estimation layer, v holds an accurate degree estimation whp.*

▶ **Lemma 7.** *When the echo process terminates, it is guaranteed that every node performed $\Omega(\log_\Delta(n))$ rounds.*

Proof. Suppose that r initiated the tree construction process at its round t. Let v be a node that maximizes $\text{dist}_G(r, v) = H$ and observe that $H = \Omega(D) = \Omega(\log_\Delta(n))$. Let t_1 be v's round at which v received a `join` message for the first time and let t_2 be v's round at which v completed its role in the echo process. We show in the full version [6] that $t_1 \geq t + H$, so also $t_2 \geq t + H$. Based on that, we conclude that if r terminates the tree construction process at its t' round, then $t' \geq t_2 + H$, hence $t' - t \geq 2H$.

Since $t > 0$, it follows that the degree estimation layer of r has made at least $2H$ rounds by the time the echo process terminates. Since $\text{dist}_G(r, u) \leq H$ for every node u, the properties of the locally synchronous scheduler ensure that the degree estimation layer of any node has made at least H rounds by that time. The assertion follows as $H = \Omega(\log_\Delta(n))$. ◀

Recall that the DAG $\widetilde{T}$ constructed by the 2-hop coloring layer may be deleted (together with the corresponding 2-hop coloring) due to a reset. In the full version [6], we address the correctness of the reset process, proving that every reset can be mapped injectively to a node that initiated a `reset_request` message and that every node that initiated such a message, performs a reset (and deletes itself from $\widetilde{T}$) within finite time. Moreover, if node v initiates a `reset_request` message over DAG $\widetilde{T}$, then, within finite time, we reach a configuration in which $\widetilde{T}$ is deleted and the network does not contain any reset messages.

4.2 Algorithm's Performance and Resources

4.2.1 Run-Time

We divide the algorithm's execution into three stages: stage (1) that lasts from the beginning of the execution until the last reset is over; stage (2) that lasts from the end of stage (1) until the echo process of the 2-hop coloring layer reaches the root; and stage (3) that lasts from the end of stage (2) until the tour construction terminates. Let T_1, T_2, and T_3 be the run-times of the first, second, and third stages, respectively.

Corollary 6 guarantees that after each node v executed $O(\log_\Delta(n))$ rounds, it holds an accurate estimation of its degree whp. This means that whp, v will not initiate a new reset after executing $O(\log_\Delta(n))$ rounds. In the full version [6], we show that a reset initiated by

some node will disappear from the network after $O(H)$ rounds of the root, where H is $\widetilde{T}$'s height. In Corollary 11, we prove that $H = O_\Delta(D)$, hence $T_1 = O_\Delta(D)$.

The second stage contains the tree construction and echo processes. The run-time of the latter is $O(H)$, which is $O_\Delta(D)$ by Corollary 11, whereas the former is intertwined with coloring trials that may delay its progression and require a more delicate analysis. This analysis is concluded in Corollary 10, showing that the run-time of the tree construction process is $O(D + \log n) = O_\Delta(D)$.

The tour construction layer is designed so that once node v completes its role in the echo process of the 2-hop coloring layer, it completes its role in the tour construction layer in $O(\Delta)$ additional rounds. Thus, the run-time of the third stage is $T_3 = O(\Delta)$. To conclude, the total run-time of our algorithm is $T_1 + T_2 + T_3 = O_\Delta(D)$.

A `color_req`$(\cdot)$ trial of node v succeeds when the following two conditions are satisfied for every node $u \in \Gamma^*(v)$: (1) the `safe` flag of u is `true`; (2) u does not witness a `color_req`$(\cdot)$ conflict in $\Gamma^*(u)$. We show that these two conditions are satisfied with probability close to 1.

▶ **Lemma 8.** *Assuming that no `reset` interrupt is signaled in $\Gamma^+(v)$, every `color_req`$(\cdot)$ trial of node v succeeds with probability $1 - O(1/\Delta)$.*

Proof. Fix some node $u \in \Gamma^*(v)$. By Observation 5, the `safe` flag of u is `true` with probability $1 - O(1/\Delta^2)$. Since the nodes use a color palette of size Δ^4 and since $|\Gamma^*(u)| \leq \Delta + 1$, we deduce by a birthday paradox argument that the probability that u witnesses a `color_req`$(\cdot)$ conflict in $\Gamma^*(u)$ is $O(1/\Delta^2)$. The assertion follows by a union bound argument over all nodes $u \in \Gamma^*(v)$. ◀

▶ **Lemma 9.** *Suppose that the root r starts a new tree construction process at round t_0 and let $\widetilde{T}$ be the constructed DAG. If node v with $\text{dist}_G(r, v) = x$ joins $\widetilde{T}$ during round t_1 of r, then whp (1) $t_1 - t_0 \leq O(x + \log n)$; and (2) the depth of v in $\widetilde{T}$ is $O(x + \log n)$.*

Proof. Let $\pi = (r = v_0, v_1, \ldots, v_x = v)$ be a shortest (r, v)-path in G. When a node is trying to pick a color, it may go through successive `color_req`$(\cdot)$ trials until one of them is approved. The actual order in which the nodes in π are colored is not necessarily identical to the order induced by π, but after x `color_req`$(\cdot)$ trails in π are approved, all nodes in π are colored.

Assuming that no `reset` interrupt is signaled in $\Gamma^+(v)$, we prove in the full version [6] that every `color_req`$(\cdot)$ trial of node succeeds with probability $1 - O(1/\Delta)$. The number of failed `color_req`$(\cdot)$ trials in π is stochastically dominated by a negative binomial random variable N with parameters x and $1-p$, where $p = 1 - O(1/\Delta) \geq \Omega(1)$ Therefore, we have to up-bound $N + x$ which accounts for the total number of `color_req`$(\cdot)$ trails in π, including the approved ones. Using standard tail bounds on the negative binomial distribution, we conclude that $N + x = O(x + \log n)$ whp.

Starting from r's round t_0, after r executed at least $(c+1)x + c \cdot \log(n) = O(x + \log n)$ rounds, it is ensured by the properties of the locally synchronous scheduler that all nodes at distance at most x from r executed at least $c(x + \log n)$ rounds. Therefore, all these nodes have fixed their color and joined $\widetilde{T}$ whp, so their depth in $\widetilde{T}$ is at most $O(x + \log n)$. ◀

▶ **Corollary 10.** *Suppose that the root r starts a new tree construction process at round t_0 after all nodes hold an accurate degree estimation. Then, this process is completed within $O(D + \log n)$ rounds of r.*

▶ **Corollary 11.** *Any DAG $\widetilde{T}$ constructed by the tree construction process satisfies* $\text{height}(\widetilde{T}) \leq O_\Delta(D)$ *whp.*

Proof. Let v be a node at distance $x \leq D$ from r. By Lemma 9, we conclude that v joins $\widetilde{T}$ in r's $t_0 + O(D + \log n)$ round whp, implying that its depth in $\widetilde{T}$ is $O(D + \log n) = O_\Delta(D)$. ◀

5 The Necessity of our Assumptions

As discussed in Sec. 1.1, from the perspective of the algorithm designer, the model considered in the current paper is weaker than that of [25] in the sense that the bounding parameter is restricted to $b = 1$ and the graph may contain self loops (cf. [5]). At the same time, the current paper makes two simplifying assumptions:
(1) the node degrees are bounded by some constant Δ; and
(2) the algorithm is provided with a unique I/O node and although this node is chosen arbitrarily, it may still serve as a (unique) *leader*, thus facilitating the design of the SSAM.
Assumption (1) is clearly mandatory for the construction of the 2-hop coloring as the number of colors is an inherent lower bound on the number of states. Whether a 2-hop coloring is indeed a prerequisite for a SSAM is left as an open question; we conjecture that the answer to this question is positive. Assumption (2) is justified by the following two lemmas.

▶ **Lemma 12.** *At the absence of a designated root node, there does not exist any SA algorithm that solves the (1-hop) coloring problem on a simple path with self-loops with failure probability bounded away from* 1*.*

▶ **Lemma 13.** *At the absence of a designated root node, there does not exist any SA algorithm that solves the* 2*-hop coloring problem on a simple path without self-loops with failure probability bounded away from* 1*.*

The proofs of Lem. 12 and 13 are based on probabilistic indistinguishability arguments, similar to those used in many distributed computing negative results, starting with the classic leader election impossibility result of Itai and Rodeh [32] (see also [5]).

Proof of Lem. 12. Our attention in this proof is restricted to algorithms operating under a fully synchronous scheduler on graph family $\{L_n^{\circlearrowleft}\}_{n \geq 1}$, where $L_n^{\circlearrowleft}$ is a simple path of n nodes augmented with self-loops. Assume by contradiction that there exists an algorithm $\mathcal{A}$ as in the lemma's statement and let Σ denote its message alphabet. Consider the execution of $\mathcal{A}$ on the instance $L_1^{\circlearrowleft}$ and let v be the (single) node in this instance. By definition, there exist a color c, constants $p > 0$ and ℓ, and message sequence $S \in \Sigma^\ell$ such that when $\mathcal{A}$ runs on this instance, with probability at least p, node v reads message $S(t)$ in its (single) port in round $t = 1, \dots, \ell$ and chooses color c at the end of round ℓ.

Now, consider graph $L_n^{\circlearrowleft}$ for some sufficiently large n whose value is determined later on and let $v_1, \dots, v_{2\ell+2}$ be any $2\ell + 2$ contiguous nodes in the underlying path of $L_n^{\circlearrowleft}$, referred to as a *gadget*. The key observation now is that when $\mathcal{A}$ runs on $L_n^{\circlearrowleft}$, with probability at least $q = p^{2\ell+2}$, both middle nodes $v_{\ell+1}$ and $v_{\ell+2}$ in the gadget receive the same message $S(t)$ in (all) their ports in round $t = 1, \dots, \ell$ and both choose color c at the end of round ℓ, independently of the random bits of the nodes outside the gadget. We refer to this event, which clearly leads to an invalid output, as a gadget *failure*. Since p and ℓ are constants that depend only on $\mathcal{A}$, $q = p^{2\ell+2}$ is also a constant that depends only on $\mathcal{A}$.

Take z to be an arbitrarily large constant. If n is sufficiently large, then we can embed $y = \lceil z/q \rceil$ disjoint gadgets in $L_n^{\circlearrowleft}$. When $\mathcal{A}$ runs on $L_n^{\circlearrowleft}$, each of these y gadgets fails (independently) with probability at least q. Therefore, the probability that $\mathcal{A}$ returns a valid output is at most $(1 - q)^y$. The assertion follows since this expression tends to 0 as $y \to \infty$ which is obtained as $z \to \infty$. ◀

The proof of Lem. 12 essentially shows that no SA algorithm can distinguish between $L_1^{\circlearrowleft}$ and $L_n^{\circlearrowleft}$ with a bounded failure probability. Regarding Lem. 13, we can use a very similar line of arguments to show that no SA algorithm can distinguish between L_2 and L_n with a bounded failure probability, thus establishing the lemma.

References

1 Y. Afek, N. Alon, O. Barad, E. Hornstein, N. Barkai, and Z. Bar-Joseph. A biological solution to a fundamental distributed computing problem. *Science*, 331(6014):183–185, 2011.
2 Yehuda Afek, Noga Alon, Ziv Bar-Joseph, Alejandro Cornejo, Bernhard Haeupler, and Fabian Kuhn. Beeping a Maximal Independent Set. In *DISC*, pages 32–50, 2011.
3 Yehuda Afek, Noga Alon, Ziv Bar-Joseph, Alejandro Cornejo, Bernhard Haeupler, and Fabian Kuhn. Beeping a Maximal Independent Set. *CoRR*, abs/1206.0150, 2012. URL: `http://arxiv.org/abs/1206.0150`, `arXiv:1206.0150`.
4 Yehuda Afek, Baruch Awerbuch, and Eli Gafni. Applying Static Network Protocols to Dynamic Networks. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, SFCS '87, pages 358–370, Washington, DC, USA, 1987. IEEE Computer Society. `doi:10.1109/SFCS.1987.7`.
5 Yehuda Afek, Yuval Emek, and Noa Kolikant. Selecting a Leader in a Network of Finite State Machines. In *DISC*, 2018. The full version can be obtained from `http://arxiv.org/abs/1805.05660`.
6 Yehuda Afek, Yuval Emek, and Noa Kolikant. The Synergy of Finite State Machines (Full Version). `http://yemek.net.technion.ac.il/files/tsfsm.pdf`, 2018.
7 D. Angluin, J. Aspnes, Z. Diamadi, M.J. Fischer, and R. Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, pages 235–253, 2006.
8 J. Aspnes and E. Ruppert. An introduction to population protocols. In Benoît Garbinato, Hugo Miranda, and Luís Rodrigues, editors, *Middleware for Network Eccentric and Mobile Applications*, pages 97–120. Springer-Verlag, 2009.
9 Baruch Awerbuch. Complexity of Network Synchronization. *J. ACM*, 32(4):804–823, 1985.
10 Y. Benenson, T. Paz-Elizur, R. Adar, E. Keinan, Z. Livneh, and E. Shapiro. Programmable and autonomous computing machine made of biomolecules. *Nature*, 414(6862):430–434, 2001.
11 S. Cannon, J.J. Daymude, D. Randall, and A.W. Richa. A Markov chain algorithm for compression in self-organizing particle systems. In *PODC*, pages 279–288, 2016.
12 Lihi Cohen, Yuval Emek, Oren Louidor, and Jara Uitto. Exploring an Infinite Space with Finite Memory Scouts. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 207–224, 2017.
13 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.
14 Alejandro Cornejo and Fabian Kuhn. Deploying Wireless Networks with Beeps. In *DISC*, pages 148–162, 2010.
15 Z. Derakhshandeh, R. Gmyr, A. Porter, A.W. Richa, C. Scheideler, and T. Strothmann. On the runtime of universal coating for programmable matter. In *DNA*, pages 148–164, 2016.
16 Z. Derakhshandeh, R. Gmyr, A.W. Richa, C. Scheideler, and T. Strothmann. An algorithmic framework for shape formation problems in self-organizing particle systems. In *NANOCOM*, pages 21:1–21:2, 2015.
17 Z. Derakhshandeh, R. Gmyr, A.W. Richa, C. Scheideler, and T. Strothmann. Universal shape formation for programmable matter. In *SPAA*, pages 289–299, 2016.

18 Z. Derakhshandeh, R. Gmyr, A.W. Richa, C. Scheideler, T. Strothmann, and S. Tzur-David. Infinite object coating in the amoebot model. *CoRR*, abs/1411.2356, 2014.

19 Z. Derakhshandeh, R. Gmyr, T. Strothmann, R.A. Bazzi, A.W. Richa, and C. Scheideler. Leader election and shape formation with self-organizing programmable matter. In *DNA*, pages 117–132, 2015.

20 S. Dolev, R. Gmyr, A.W. Richa, and C. Scheideler. Ameba-inspired self-organizing particle systems. *CoRR*, abs/1307.4259, 2013.

21 D. Doty. Timing in chemical reaction networks. In *SODA*, pages 772–784, 2014.

22 Y. Emek, T. Langner, D. Stolz, J. Uitto, and R. Wattenhofer. How many ants does it take to find the food? *Theor. Comput. Sci.*, 608:255–267, 2015.

23 Y. Emek, T. Langner, J. Uitto, and R. Wattenhofer. Solving the ANTS problem with asynchronous finite state machines. In *ICALP*, pages 471–482, 2014.

24 Y. Emek and J. Uitto. Dynamic Networks of Finite State Machines. In *SIROCCO*, pages 19–34, 2016.

25 Y. Emek and R. Wattenhofer. Stone age distributed computing. In *PODC*, pages 137–146, 2013. The full version can be obtained from `http://yemek.net.technion.ac.il/files/stone-age.pdf`.

26 Yuval Emek, Tobias Langner, David Stolz, Jara Uitto, and Roger Wattenhofer. How Many Ants Does It Take To Find the Food? In *21th International Colloquium on Structural Information and Communication Complexity (SIROCCO), Hida Takayama, Japan*, July 2014.

27 Yuval Emek, Christoph Pfister, Jochen Seidel, and Roger Wattenhofer. Anonymous networks: randomization = 2-hop coloring. In *ACM Symposium on Principles of Distributed Computing, PODC*, pages 96–105, 2014.

28 O. Feinerman and A. Korman. Memory lower bounds for randomized collaborative search and implications for biology. In *DISC*, pages 61–75, 2012.

29 O. Feinerman and A. Korman. *Theoretical distributed computing meets biology: a review*, pages 1–18. Springer Berlin Heidelberg, 2013.

30 O. Feinerman, A. Korman, Z. Lotker, and J.S. Sereni. Collaborative search on the plane without communication. In *PODC*, pages 77–86, 2012.

31 M. Gardner. The fantastic combinations of John Conway's new solitaire game 'life'. *Scientific American*, 223(4):120–123, 1970.

32 Alon Itai and Michael Rodeh. Symmetry breaking in distributed networks. *Inf. Comput.*, 88(1):60–87, 1990.

33 T. Langner, B. Keller, J. Uitto, and R. Wattenhofer. Overcoming obstacles with ants. In *OPODIS*, pages 9:1–9:17, 2015.

34 T. Langner, J. Uitto, D. Stolz, and R. Wattenhofer. Fault-tolerant ANTS. In *DISC*, pages 31–45, 2014.

35 O. Michail, I. Chatzigiannakis, and P.G. Spirakis. *New models for population protocols*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2011.

36 S. Navlakha and Z. Bar-Joseph. Distributed information processing in biological and computational systems. *Commun. ACM*, 58(1):94–102, 2014.

37 David Peleg. *Distributed Computing: A Locality-sensitive Approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

38 NSF workshop on self-organizing particle systems (SOPS). `http://sops2014.cs.upb.de/`, 2014.

39 J. von Neumann. *Theory of self-reproducing automata*. University of Illinois Press, Champaign, IL, USA, 1966.

40 S. Wolfram. *A new kind of science*. Wolfram Media, Champaign, Illinois, 2002.

Task Computability in Unreliable Anonymous Networks

Petr Kuznetsov
LTCI, Télécom ParisTec, Université Paris-Saclay, France
petr.kuznetsov@telecom-paristech.fr

Nayuta Yanagisawa
DeNA Co., Ltd., Japan
nayuta.yanagisawa@dena.com

Abstract

We consider the *anonymous broadcast* model: a set of n anonymous processes communicate via *send-to-all* primitives. We assume that underlying communication channels are asynchronous but reliable, and that the processes are subject to *crash* failures. We show first that in this model, even a single faulty process precludes implementations of *atomic* objects with non-commuting operations, even as simple as read-write registers or add-only sets. We, however, show that a *sequentially consistent* read-write memory and add-only sets can be implemented t*-resiliently* for $t < n/2$, i.e., provided that a majority of the processes do not fail. We use this implementation to establish an equivalence between the t-resilient read-write anonymous shared-memory model and the t-resilient anonymous broadcast model in terms of colorless task solvability. As a result, we obtain the first task computability characterization for *unreliable* anonymous message-passing systems.

2012 ACM Subject Classification Theory of computation → Distributed computing models, Theory of computation → Computability

Keywords and phrases Distributed tasks, anonymous broadcast, fault-tolerance

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.23

1 Introduction

Algorithms for conventional distributed systems assume that every process is assigned a distinct identifier that can be used in the code. However, *anonymous* algorithms that have to program processes identically can be used in a much wider context. Many systems, such as sensor networks, web services, peer-to-peer file repositories, are anonymous by design or may choose anonymity for the sake of users' privacy [2, 10].

In this paper, we consider computability issues of anonymous systems. This topic is traditionally tackled in *shared-memory* models where a set of asynchronous (or partially synchronous) anonymous processes communicate via *multi-writer multi-reader* shared memory locations. A number of interesting complexity and computability bounds [16, 7, 12, 20, 8, 6] have been established for these models. Guerraoui and Ruppert [16] showed that many interesting abstractions, such as timestamps and atomic snapshots, can be implemented in a *wait-free* way, i.e., tolerating an arbitrary number of faulty processes. Yanagisawa [20] characterized the class of *colorless tasks* that can be solved wait-free in this model.[1] It turns

[1] Informally, a colorless task, such as consensus or set agreement, is defined in terms of relations between *sets* of inputs and *sets* of outputs, so they allow natural formulations for anonymous systems. A wait-free algorithm guarantees that a process *makes progress*, e.g., produces a task output, in a finite number of its own steps, even if all $n-1$ remaining processes are faulty.

© Petr Kuznetsov and Nayuta Yanagisawa;
licensed under Creative Commons License CC-BY
22nd International Conference on Principles of Distributed Systems (OPODIS 2018).
Editors: Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira; Article No. 23; pp. 23:1–23:13

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

out the class is precisely the tasks that can be solved in the conventional *non-anonymous* model, i.e., getting rid of process identifiers does not make the system weaker with respect to solving colorless tasks. The equivalence has been recently generalized to t-resilient models assuming that at most t processes are allowed to fail by Delporte et al. [13]: a colorless task is t-resiliently solvable in the (non-anonymous) read-write shared-memory model if and only it is t-resiliently solvable in the anonymous counterpart.

Anonymous computing in networks. One can argue that shared memory may be an inadequate communication model for *large-scale* systems, which typically motivate assuming anonymous algorithms. In this paper, we propose to have a closer look at anonymous computations in *message-passing* systems. More precisely, we consider systems in which processes communicate via the reliable *anonymous broadcast* primitive: every broadcast message is eventually received and the integrity of received messages is ensured, while it is impossible to detect the source of a given message.

The model has been considered earlier in the *fault-free* context by Aspnes et al. [3]. They showed that if no process can fail, any *idemdicent* object can be implemented in the anonymous broadcast model in the *linearizable* way, i.e., ensuring that each operation on it appears to be executed atomically within the operation's interval. Intuitively, for an idemdicent object, such as a *register*, an *add-only set*, or a *counter* with separate increment and read operations, any operation modifying the object's state returns only a (non-informative) *ack* response.

Unreliable anonymous networks. We propose a set of computability results for the anonymous broadcast model with *unreliable* processes. We show first that no object exporting non-commuting operations, including registers and add-only sets, has a linearizable implementation in this model, as long as just a single process can fail. We leverage the fact that, in anonymous message-passing systems, no process can detect whether a given message comes in reaction to its own action or to an earlier action performed by its *clone*, i.e., a process having an identical state. Therefore, it might become impossible to ensure that operations on the implemented object appear in an order that preserves their real-time precedence, which is required by linearizability.

We show, however, that a t-resilient *sequentially consistent* add-only set can be implemented in the anonymous broadcast model, assuming that $t < n/2$, i.e., less than half of the processes are allowed to fail. Unlike linearizable objects, sequentially consistent ones do not ensure that the sequential execution preserves the real time precedence between operations and, as a result, sequential consistency is not a *composable* property [17]. Our t-resilient implementation of an add-only set, inspired by the wait-free atomic snapshot algorithm by Afek et al. [1], is interesting in its own right. To ensure that the views of the set contents evaluated by different processes are consistent with the same sequential execution, a *get* operation is only allowed to return a set of values that is agreed upon by a majority of processes. To guarantee that every correct process completes each of its operations, we introduce a *helping* mechanism: every value added to the set is equipped with a view that can be used by future *get* operations.

Colorless tasks in unreliable anonymous networks. Further, we show that a sequentially consistent add-only set is as good as a linearizable one, as long as colorless task computability is concerned. More precisely, we can characterize the class of colorless tasks that are t-resiliently solvable in the anonymous broadcast model, where $t < \frac{n}{2}$. The characterization

stipulates that a colorless task is t-resilient solvable in the *anonymous broadcast* model if and only if it is t-resiliently solvable in the anonymous *shared-memory* model, where $t < \frac{n}{2}$. Thus, for colorless tasks, the anonymous broadcast model with a majority of correct processes is as powerful as the anonymous shared-memory model and, by recalling the equivalence of Delporte et al. [13], the regular (non-anonymous) shared-memory model.

Technically, the *if* part of our characterization is established by the following two steps: First, we present an implementation of a *sequentially-consistent* snapshot. Then, we show that any algorithm that t-resiliently solves a colorless task using an atomic snapshot memory also solves the colorless task with a sequentially-consistent snapshot memory. The *only if* part follows from the characterization of the t-resilient colorless task solvability in the anonymous *shared-memory* model by Delporte et al. [13].

Related work. Colorless tasks, a fundamental class of distributed problems, include consensus [15], set agreement [9], and loop agreement [19]. The class was extensively studied mainly in the context of the non-anonymous shared-memory computing [18]. There are recent papers that have studied the computability of colorless tasks in the anonymous shared-memory model [20, 13].

In the non-anonymous setting, the message-passing model and the shared-memory model are equivalent in the sense of simulation [4], where $t < \frac{n}{2}$. Thus, a colorless task can be solved in the shared-memory model if and only if it can be solved in the t-resilient message-passing model. The situation is different in the anonymous case. In general, the anonymous broadcast model and the anonymous shared-memory model cannot simulate each other. Thus, results concerning the anonymous shared-memory colorless task computability cannot be directly applied to the anonymous broadcast model.

The add-only set object has been first introduced under the name of *weak set* by Delporte and Fauconnier [11]. The object has been studied mainly in the context of anonymous shared-memory computing and used to characterize anonymous shared-memory systems concerning colorless tasks [20, 13, 14].

Failure detectors that enable solutions to the consensus task in unreliable anonymous networks have been discussed by Bonnet and Raynal [5]. In contrast, this paper focuses on general task computability in asynchronous anonymous networks, i.e., without help from external oracles.

Roadmap. The rest of the paper is organized as follows. In Section 2, we give our model definitions. In Section 3, we show that no type with weakly non-commutative operations allows a linearizable implementation in an unreliable atomic broadcast model. In Section 4, we present a sequentially consistent implementation of an add-only set in the anonymous broadcast model with a majority of correct processes. In Section 5, we present our anonymous colorless task computability theorem. In Section 6, we discuss relaxations of linearizability that might fit the anonymous broadcast context.

2 Preliminaries

We consider the *anonymous* model of n processes, $p_1, \ldots, p_n$, that have no knowledge of their identifiers and execute an identical algorithm. We particularly focus on the *anonymous asynchronous broadcast model*. In the model, a distributed system is composed of n anonymous processes that run asynchronously and communicate by broadcasting messages via a fully connected reliable network: every pair of processes are connected via a first-in-first-out

(FIFO) reliable link. To send a message m to all, a process invokes *broadcast*(m) primitive, and an event *receive*(m) occurs when the message is received.

Broadcast models. Here we distinguish two broadcast models: *instantaneous* and *non-instantaneous* broadcasts. In the instantaneous broadcast model, every event *broadcast*(m) performed by a process p_i instantaneously sends message m to every other process and, as the links are assumed to be reliable, it is guaranteed that the message will eventually be received by every correct process. In the non-instantaneous broadcast model, if *broadcast*(m) is the last event performed by a *faulty* process, then it sends m to a *subset* of processes and guarantees that the message will be eventually received by every correct process *in this subset.* In both cases, the communication channels ensure FIFO semantics: the messages are delivered in the order they have been sent. Notice, however, that a process cannot detect through which link a message has been received.

Our impossibility result is shown for the stronger instantaneous model and our upper bounds hold for the weaker non-instantaneous model. (By default we assume that the model is non-instantaneous.)

Object types and implementations. A sequential object type is defined as a tuple $T = (Q, q_0, O, R, \Delta)$, where Q is a set of states, $q_0 \in Q$ is an initial state, O is a set of operations, R is a set responses and $\Delta \subseteq Q \times O \times Q \times R$ is a relation that associates a state and an operation to a set of possible new states and corresponding responses. Here we assume that Δ is total on the first two elements, i.e., for each state $q \in Q$ and each operation in $o \in O$, some transition to a new state is defined, i.e., $\forall (q, o) \in Q \times O, \exists (q', r) \in Q \times R$: $(q, o, q', r) \in \Delta$.

A *history* is a sequence of (operation) invocations and responses and a sequential history is a history that starts with an invocation of an operation and in which every invocation is immediately followed with a *matching* response. A *sequential history* $o_1, r_1, o_2, r_2, \ldots$, where $\forall i \geq 1, o_i \in O,\ r_i \in R$, is *legal* with respect to type $T = (Q, q_0, O, R, \Delta)$ if there exists a sequence $q_1, q_2, \ldots$ of states in Q such that $\forall i \geq 1$, $(q_{i-1}, o_i, q_i, r_i) \in \Delta$.

An *implementation* of an object type T is an algorithm that, for each invoked operation, prescribes the actions that a process needs to take to perform it. In our case, the actions are invocations of *broadcast* primitives, processing of *receive* events and returning responses to the invoked operations. An *execution* of an implementation is a sequence of *events*: invocations and responses of operations, *broadcast* calls and *receive* events: the sequence of events at every process must respect the algorithm assigned to it. A process is called *faulty* in an infinite execution if it stops before performing an event prescribed by its algorithm; otherwise it is called *correct*.

Linearizability and sequential consistency. We assume that no process invokes a new operation before receiving a response for the previous one. For each pattern of invocations, the implementation produces a *history*, i.e., the sequence of distinct invocations and responses, labelled with process identifiers and unique sequence numbers.

A projection of a history H to process p_i, denoted $H|i$ is the subsequence of elements of H labelled with p_i. An invocation o by a process p_i is *incomplete* in H if it is not followed by a response in $H|i$. A history is *complete* if it has no incomplete invocations. A *completion* of H is a history $\bar{H}$ that is identical to H except that every incomplete invocation in H is either removed or *completed* by inserting a matching response somewhere after it.

A *sequentially consistent* implementation of an object type T ensures that for every history H it produces, there exists a completion $\bar{H}$ and a legal sequential history S such that for all processes p_i, $\bar{H}|i = S|i$.

A *linearizable* implementation, additionally, preserves the real-time order between invocations. Formally, an invocation o_1 *precedes* an invocation o_2 in H, denoted $o_1 \prec_H o_2$, if o_1 is complete and the corresponding response r_1 precedes o_2 in H. Note that $\prec_H$ stipulates a partial order on invocations in H. Now a linearizable implementation of T ensures that for every history H it produces, there exists a completion $\bar{H}$ and a legal sequential history S such that (1) for all processes p_i, $\bar{H}|i = S|i$ and (2) $\prec_H \subseteq \prec_S$.

A (sequentially consistent or linearizable) implementation is *t-resilient* if, under the assumption that at most t processes crash, it ensures that every invocation performed by a correct process is eventually followed by a response.

An *anonymous* implementation is not allowed to use process identifiers: every process is assigned the same algorithm that only depends on the sequence of operation invocations and received messages.

3 Linearizable objects with non-commuting operations

In this section, we show that nontrivial sequential object types cannot be implemented in a linearizable way in the anonymous broadcast model if $t \geq 1$, i.e., at least one process may fail.

Given a type $T = (Q, q_0, O, R, \Delta)$, we say that operations $o_1, o_2 \in O$ are *weakly-non-commutative* if, for all $r_1, r_2 \in R$ such that $o_1 r_1 o_2 r_2$ is legal, $o_1 r_1 o_2 r_2 o_1 r_1$ is not legal. Intuitively, o_1 is a read operation and o_2 is an update: swapping the order of the two operations affects the response of o_1.

Two examples of types with weakly non-commutative operations are *read-write register* and *add-only set*. A register stores an integer value, initially 0, and exports operations *read*, which returns the value, and *write*(v), $v \in \mathbb{N}$, which replaces the value with v. Here *read*() and *write*(1) are examples of weakly non-commutative operations. An add-only set stores a set of integer values, initially $\emptyset$, and exports operations *get*, which returns the set, and *add*(v), $v \in \mathbb{N}$, which adds v to the set. Similarly, *get*() and *add*(1) are weakly non-commutative.

▶ **Theorem 1.** *There does not exist a 1-resilient linearizable implementation of a type with weakly non-commutative operations in the anonymous instantaneous broadcast model.*

Proof. Suppose, by contradiction, that there exists such an implementation of a type $T = (Q, q_0, O, R, \Delta)$ with weakly non-commutative operations. Let $o_1, o_2 \in O$ be operations of T such that for all $r_1, r_2 \in R$, whenever $o_1 r_1 o_2 r_2$ is legal, it holds that $o_1 r_1 o_2 r_2 o_1 r_1$ is not legal.

Consider the following execution of the implementation. Process p_1 invokes o_1 and takes steps of the algorithm until a response r_1 is returned, then invokes o_2 and takes steps of the algorithm until a response r_2 is returned. In this execution, we pick another process p_2 and delay all messages broadcast *by and to* p_2 at least until r_2 is returned at p_1. The remaining processes obediently take steps of the algorithm and all messages sent across these processes are eventually received. Note that both o_1 and o_2 must eventually return at p_1, as the implementation is 1-resilient and, from p_1's perspective, the execution is indistinguishable from an execution in which p_2 has crashed initially and all other processes are correct. Let α be the resulting (finite) execution and $e_1, \ldots, e_k$ be the subsequence of events of p_1 in α that starts with o_1 and ends with r_1.

Let $e'_1, \ldots, e'_k$ be the sequence of events that is identical to $e_1, \ldots, e_k$, except that every event is now labelled with p_2 (or, in other words, is performed by p_2).

We claim that $\alpha e'_1, \ldots, e'_k$ is an execution of our implementation. By the construction of α, e'_1 is the invocation of o_1, i.e., $\alpha e'_1$ is an execution of the implementation. Inductively,

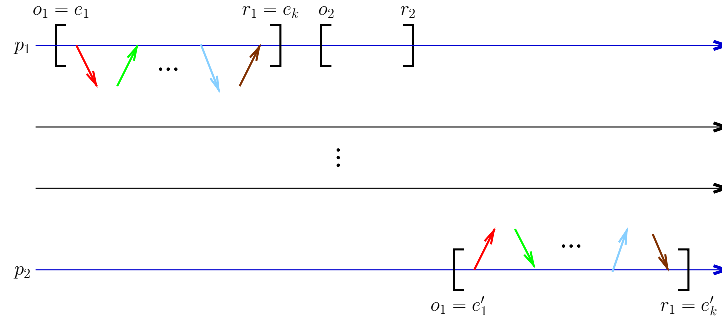


Figure 1 The execution constructed in the proof of Theorem 1. Processes p_1 and p_2 are clones: they execute exactly the same sequence of events in which operation o_1 returns response r_1.

suppose that for some ℓ, $1 \leq \ell < k$, $\alpha e'_1, \ldots, e'_\ell$ is an execution of the implementation. Note that, since the algorithm is anonymous, the local state of p_2 after $\alpha e'_1, \ldots, e'_\ell$ is identical to the local state of p_1 after $e_1, \ldots, e_\ell$.

Thus, if $e_{\ell+1}$ is a broadcast event, then p_2 can also perform a similar broadcast event $e'_{\ell+1}$ after $\alpha e'_1, \ldots, e'_\ell$ and, thus, $e'_{\ell+1}$ after $\alpha e'_1, \ldots, e'_{\ell+1}$ is indeed an execution.

Now suppose that $e_{\ell+1}$ is a receive event, for some message m previously broadcast by a process p_i. Recall that in α, p_2 has not received a single broadcast message. Note that in $\alpha e'_1, \ldots, e'_\ell$, p_2 received every message received by p_1 in $e_1, \ldots, e_\ell$, but no other messages. Thus, as m is the next message to be received by p_1 from p_i in $e_1, \ldots, e_\ell$, it is also the next message to be received by p_2 from p_i in $\alpha e'_1, \ldots, e'_\ell$. Hence, $\alpha e'_1, \ldots, e'_{\ell+1}$ is indeed an execution of our implementation. By induction, we constructed an execution $\alpha e'_1, \ldots, e'_k$ which produces a history $o_1 r_1 o_2 r_2 o_1 r_1$ that is not legal – a contradiction. ◀

4 Sequentially consistent add-only set

We now describe a sequentially consistent t-resilient implementation of *add-only set* in the anonymous broadcast model.

Overview. In spirit, our t-resilient implementation (Algorithm 1) is close to the wait-free atomic snapshot implementation by Afek et al. [1]. Similar to the snapshot operation in the algorithm by Afek al., a *get* operation executed by p_i repeatedly broadcasts its current (constantly evolving) evaluation of the set contents until either "sufficiently many" processes agree with it, or some process p_j reports a set of values containing all the values known to p_i at the beginning of the operation (in which case p_j helps p_i with completing the operation).

The algorithm proceeds in monotonically increasing *rounds*. Every process p_i maintains a set of values V_i, its local version of the set contents, where each value is equipped with a "witness set" (the set obtained by a *get* operation before the value has been added).

A *get* operation returns a set of values as soon as it establishes that a majority of processes evaluate the same set of values *for the current round* (line 21). Here, given a set of tuples $[u, W]$, where u is a value and W is a set of values "witnessed" by the process that added u, *values*(V) returns the set of first entries of tuples in V.

Otherwise, the process updates its local set with new values and proceeds to the next round. Note that if no values are added to the set from some point on, every get operation

Algorithm 1 Implementation of sequentially consistent add-only set: code for process p_i

```
1  Local variables:
2      V_i: set initially ∅          // local estimate of the set of values,
3                                    // equipped with witness sets
4      M_i^r: multiset initially ∅   // messages of round r
5      r_i: integer initially 0      // current round number

6  add(v):
7      Vals = get()
8      V_i = V_i ∪ {[v, Vals]}
9      broadcast([V_i])

10 upon receive [V]
11     V_i = V_i ∪ V

12 get():
13     U = V_i
14     while true do
15         r_i++
16         V = V_i
17         if M_i^{r_i} = ∅ then
18             broadcast([V, r_i])
19         wait until |M_i^{r_i}| > n/2
20         if ∀V' ∈ M_i^{r_i} : V' = V then
21             return values(V)
22         if ∃V' ∈ M_i^{r_i}, ∃[u, W] ∈ V' : values(U) ⊆ W then
23             return W

24 upon receive [V, r]
25     V_i = V_i ∪ V
26     if M_i^r = ∅ then
27         broadcast([V_i, r])
28     M_i^r = M_i^r ∪ {V}
```

invoked by a correct process will eventually return. Thus, the only reason for a get operation not to return is an infinite number of successful add operations performed by a concurrent process. To ensure that each correct process eventually completes every operation it invokes, we introduce a *helping* mechanism: before adding a new value to the set, a process *gets* the current set of values (line 7) and attaches it to the new value. If, within a *get* operation, a process finds a value equipped with a "sufficiently recent" outcome of another *get* operation, it adopts it and outputs as its own (line 23).

Of course, in an anonymous system, detecting the source of a received message is impossible. Therefore, to guarantee that all received messages associated with a round r are coming from distinct processes and to render the "majority condition" in line 19 meaningful, we require that a process broadcasts a message associated with a given round exactly once (lines 25-27). Of course, such messages can be associated to an identical request sent by a process different from p_i (clone of p_i). But, as we will see below, this is not an issue for the algorithm's correctness. To account for a "slow" process that might receive messages associated with rounds the process has not reached yet, it proactively stores all the messages it receives.

Correctness. Fix an execution of Algorithm 1. We show first that *views*, sets of values returned by *get* operations, are ordered by containment.

▶ **Lemma 2.** *For any two views U and W returned by get operations, $U \subseteq W$ or $W \subseteq U$.*

Proof. Note that each set of values U returned by a *get* operation is either evaluated by the operation itself and returned in line 21 or "adopted" from another *get* operation that terminated earlier and returned in line 23. But there can be only finitely many *get* operations that terminated before a given *get* operation returns. Thus, each such view U was returned in line 21 by some *get* operation. We then can associate U with the round number r in which this happened, we say U *was returned* in round r.

Let U and W be returned, respectively in rounds r and r', $r \leq r'$. We show below that $U \subseteq W$. Indeed, a *get* operation returns U in round r if a majority of processes broadcast the same message $[V, r]$ such that $values(V) = U$. Since a process only broadcasts its set for a given round only once, two processes returning in a given round return the same set of values. Furthermore, if a process returns U in round r, then every process p_i that passed through that round, will get $U \subseteq values(V_i)$. Thus, W returned in round $r' \geq r$ must satisfy $U \subseteq W$. ◀

▶ **Lemma 3.** *If two* get *operations invoked by the same process p_i return U and then W, then $U \subseteq W$.*

Proof. By the algorithm, every view returned by a *get* operations (lines 13 and 22) invoked by a process p_i contains $values(V_i)$ at the time of the invocation. As V_i grows monotonically with time (lines 8 and 25), we have $U \subseteq W$. ◀

▶ **Lemma 4.** *If p_i complete an $add(v)$ and later a get operation that returns U, then $v \in U$.*

Proof. As every *get* operation by p_i returns a superset of $values(V_i)$ evaluated at the moment of its invocation (line 13), and every $add(v)$ operation by p_i adds v to $values(V_i)$ (line 13), and V_i grows monotonically with time (lines 8 and 25), we have $v \in U$. ◀

▶ **Lemma 5.** *Every operation invoked by a correct process eventually returns.*

Proof. Suppose, by contradiction, that an operation *op* invoked by a correct process p_i never terminates. Note that *op* must be a *get* operation, either invoked directly or within an *add* operation. By the algorithm, as at most $t < n/2$ processes are faulty, p_i goes through infinitely many rounds in lines 14–23. Let U be the set of values (line 13) with which p_i started *op*.

Suppose first that there is a time after which no process completes an *add* operation. Then, there is a time after which no process p_j adds a new value to its local V_j. Since the broadcast channels are reliable, every message sent by a correct process is eventually received by every correct process (line 27). Thus, there exists a round r and a set of tuples V such that every process that broadcasts a message of the kind $[V', r']$, where $r' \geq r$, will have $V' = V$. Eventually, after sufficiently many rounds, p_i will reach a round for which every received set of values is V and, thus, p_i will return $values(V)$ in line 23 – a contradiction.

Thus, a concurrent process p_j completes infinitely many *add* operations. Before adding a new value v, p_j performs a *get* operation and attaches the returned view to the added value v. Moreover, eventually p_j will receive and include in V_j all values that appear in U. Thus, eventually, p_j will attach a view W such that $values(U) \subseteq W$ to every value it adds (line 8). As p_j is correct, $[v, W]$ will be included to V_i and p_i will return in line 23 – a contradiction. ◀

▶ **Theorem 6.** *Algorithm 1 implements a sequentially consistent add-only set in the anonymous non-instantaneous broadcast model.*

Proof. Fix an execution of the algorithm. Let H be the corresponding history.

By Lemma 2, all views returned by complete *get* operations in this execution can be totally ordered based on the growing containment relation. Let S be the corresponding sequential history of *get* operations. By the algorithm, all values that appear in these views are arguments of some (complete or incomplete) *add* operations. Thus, we can amend S by inserting each such operation $add(v)$ just before the first *get* in S that returns a view containing v. By Lemma 3 and 4, we can choose S to be consistent with local histories $H|i$ which end up with complete *get* operations.

If there are only finitely many complete *get* operations in H, all complete *add* operations in H whose arguments do not appear in views returned in H can be inserted after the last *get* operation in S without affecting legality.

Otherwise, if there are infinitely many *get* operations in H, we claim that the argument of every complete *add* operation in H appears in some view returned in H. Indeed, by the algorithm (line 9), every complete $add(v)$ operation by p_i broadcasts V_i that includes v and, eventually, every correct process will have v in its local view. Thus, there is a time after which every complete *get* operation returns a view containing v.

Thus, the resulting legal sequential history S is equivalent to a completion of H that contains all complete *add* and *get* operations and some incomplete *add* operations in H.

Finally, by Lemma 5, every operation invoked by a correct process returns. Thus, Algorithm 1 is a sequentially consistent implementation of add-only set. ◀

5 Colorless tasks in anonymous networks

Our sequentially consistent implementation of an add-only set allows us to prove several interesting computability results for *colorless tasks*.

Colorless tasks. A process invokes a distributed *task* with an *input* value and the task returns an *output* value, so that the inputs and the outputs across the processes respect the task specification. In a *colorless* task, processes are free to use each others' input and output values, so the task can be defined in terms of input sets and output sets.

Formally, a *colorless task* is defined through a set $\mathcal{I}$ of input sets, a set $\mathcal{O}$ of output sets, and a total relation $\Delta : \mathcal{I} \mapsto 2^{\mathcal{O}}$ that associates each input set with a set of possible output sets. We require that Δ is *carrier* map i.e., $\forall \tau, \sigma \in \mathcal{I} : \tau \subseteq \sigma \Rightarrow \Delta(\tau) \subseteq \Delta(\sigma)$. A colorless task is said to be *t-resiliently solvable* if there exists a t-resilient protocol that guarantees that, in every execution with t or fewer failures and input set $\sigma \in \mathcal{I}$, every correct process outputs a value such that the output set τ satisfies $\tau \in \Delta(\sigma)$. Check [18] for more details on the definition.

In this section, we show that the anonymous broadcast model is equivalent, from the colorless task computability viewpoint, to the non-anonymous read-write shared memory model.

Sequentially-consistent snapshot. We now describe an implementation of a *sequentially-consistent* snapshot object, that maintains a vector of ℓ values and exports two types of operations, `update`(i, v) and `snapshot`$()$, where $i = 0, 1, \dots, \ell$, with the following sequential specification. Operation `update`(i, v) replaces the content of the i-th component of the object with value v, and `snapshot`$()$ returns the current state of the vector. Without loss of

Algorithm 2 implementation of sequentially-consistent snapshot: code for a process p_i

```
1  Shared variables:
2      SET: a sequentially-consistent add-only set

3  update(i,v):
4      S = SET.get()
5      t = maxTS(S)
6      SET.add([i,t+1,v])

7  snapshot():
8      S = SET.get()
9      v = recent(S)
10     return v

11 macro maxTS(S):
12     if S = ∅ then
13         return 0
14     return max{t | ∃i,v : [i,t,v] ∈ S}

15 macro recent(S):
16     for i in 1,...,ℓ do
17         snap[i] = recent_i(S)
18     return snap

19 macro recent_i(S):
20     if {(t,v) | ∃i : [i,t,v] ∈ S} = ∅ then
21         return ⊥
22     (t',v') = max{(t,v) | ∃i : [i,t,v] ∈ S}
23     return v'
```

generality, we assume that the values that can be stored in the vector are natural numbers. We also assume that each component of the object is initialized with a default value $\bot$.

Our implementation of a sequentially-consistent snapshot maintains a single shared sequentially consistent set `SET` and operates as follows. To update the i-th component by a value v, a process takes the current copy of the underlying add-only set, calculates the largest timestamp t, increments it, and then adds tuple $[i, t, v]$ to the set. To take a snapshot, the process first evaluates the current state S of the underlying add-only set. Then, the process calculates the most recent value of the each i-th component based on the lexicographic order on the set $\{(t, v) \mid [i, t, v] \in S\}$, where the empty set corresponds the initial value $\bot$. Overall, the implementation, presented in Algorithm 2, resembles an implementation of a linearizable register in the non-anonymous broadcast model [4].

▶ **Theorem 7.** *Algorithm 2 implements a sequentially-consistent snapshot in the anonymous non-instantaneous broadcast model.*

Proof. Fix an execution of the algorithm and let H be the corresponding history. In the executions, each `update`(i, v) is associated with a timestamp $\mathrm{TS}(\texttt{update}(i, v)) = \texttt{maxTS}(S)+1$ for S in Line 4. For each `snapshot`(), underlyingSet(`snapshot`()) denotes the set S of Line 8. We write $(t, v) < (t', v')$ if and only if (t, v) is lexicographically smaller than (t', v').

We may assume, without loss of generality, that every process performs updates and snapshots alternatively, e.g., running a full-information protocol: the first update carries the input value of the process and every next update carries the outcome of the preceding snapshot operation.

We can now construct a sequential history S corresponding to H in the following manner: Let op_1 and op_2 be update or snapshot operations appeared in H.

- In the case of $op_1 = \texttt{update}(i, v)$ and $op_2 = \texttt{update}(i, v')$, op_1 precedes op_2 in S if and only if $(\mathrm{TS}(op_1), v) < (\mathrm{TS}(op_2), v')$.
- In the case of $op_1 = \texttt{snapshot}()$ and $op_2 = \texttt{snapshot}()$, op_1 precedes op_2 in S if and only if $\mathrm{underlyingSet}(op_1) \subseteq \mathrm{underlyingSet}(op_2)$.
- In the case of $op_1 = \texttt{update}(i, v)$ and $op_2 = \texttt{snapshot}()$, op_1 precedes op_2 in S if and only if the tuple $[i, t, v]$ that op_1 has added at Line 6 is in $\mathrm{underlyingSet}(op_2)$.

In all other cases, op_1 and op_2 can be arbitrarily ordered as long as the order keeps the process local histories of H. ◀

Since decision tasks do not concern the order of inputs or outputs, the following lemma holds:

▶ **Lemma 8.** *Assume that there is an anonymous shared-memory protocol that runs on top of an atomic snapshot object and t-resiliently solve a colorless task T. Then the very same protocol can be run on top of a sequentially-consistent snapshot object and still t-resiliently solve the colorless task T.*

Proof. Let P (resp. P') be the protocol that runs on top of the atomic snapshot object (resp. a sequentially-consistent object). Fix a set of inputs $\sigma \in \mathcal{I}$ and fix the execution E' of the protocol P' starting from σ. Let H' be the corresponding history of E' and S' be the sequential history that is equivalent to H'. Then, there exists an execution E of the protocol P that corresponds to the execution S'.

In executions E and E', every process p_i reads and writes the exactly same values in the same order. Thus, if the process terminates and makes output in E, the process must terminate and produce the same output in E'. By the above arguments, if the protocol P t-resiliently solves the given colorless task T, the protocol P' also t-resiliently solves the colorless task T. ◀

Therefore, every decision task that can be t-resiliently solved with an atomic snapshot object can be t-resiliently solved with a sequentially-consistent snapshot object.

Combining Theorem 6, Theorem 7, and Lemma 8 we obtain the following result.

▶ **Lemma 9.** *Every colorless task that is t-resiliently solvable in the anonymous shared-memory model is also t-resiliently solvable in the anonymous broadcast model, where $t < \frac{n}{2}$.*

We can prove the converse of Lemma 9 by [13, Theorem 3].

▶ **Theorem 10** ([13, Theorem 3])**.** *A colorless task is t-resiliently solvable in the anonymous shared-memory model if and only if it is t-resiliently solvable in the non-anonymous one.*

▶ **Lemma 11.** *Every colorless task that is t-resiliently solvable in the anonymous broadcast model is also t-resiliently solvable in the anonymous shared-memory model, where $t < \frac{n}{2}$.*

Proof. If a colorless task T is t-resiliently solvable in the anonymous broadcast model, it is obviously t-resiliently solvable in the non-anonymous broadcast model. Then, T is also t-resiliently solvable in the non-anonymous shared-memory model because the non-anonymous shared-memory model simulates the non-anonymous broadcast model, where $t < \frac{n}{2}$ [4]. Thus, by Theorem 10, the colorless task T is t-resiliently solvable in the anonymous shared-memory model. ◀

Let us note that the above argument is necessary because, in the anonymous setting, the shared-memory model does not simulate the broadcast model. In the anonymous shared-memory model, processes cannot know the number of clone processes if they take steps alternately and keep in the identical state. On the other hand, in the anonymous broadcast model, each process eventually receives the messages sent by clone processes by an amount equal to the number of the clones.

▶ **Theorem 12.** *A colorless task T is t-resiliently solvable in the anonymous broadcast model if and only if it is t-resiliently solvable in the anonymous shared-memory model, where $t < \frac{n}{2}$.*

The theorem says that the anonymous broadcast model is equivalent to the anonymous shared-memory model from the viewpoint of colorless task solvability when less than majority of processes may fail.

6 Concluding remarks: on linearizability in anonymous networks

We showed that the anonymous broadcast model does not allow *linearizable* implementations of nontrivial object types (Theorem 1). On the positive side, we suggested to consider sequential consistency and proposed a sequentially consistency implementation of *add-only set*. While sequential consistency is good enough for exploring *one-shot* task computability (Section 5), it may not be an attractive property for *long-lived* abstractions that can be *composed* in a larger context. Unlike linearizability, the property does not preserve *real-time precedence* of operations: precisely because of this, sequentially consistent implementations do not compose [17].

A closer look at the proof of Theorem 1 reveals that sequential consistency might be a very rough fix to obviate the impossibility. Indeed, in the constructed non-linearizable history, a clone of a process that completed its operation earlier cannot distinguish the current state from the initial one and, thus, must return a "stale", inconsistent response. However, *modulo* operations executed by the clone, the constructed history can be seen as a linearizable one that preserves real-time ordering of "original" (non-cloning) operations.

Intuitively, we can think of a straightforward modification of our *add-only set* implementation that seems to enable this kind of relaxed linearizability. Before returning, an *add* operation may ensure that the added value is accepted by a majority of processes.

An interesting open question is whether this intuition is justified: can we come up with a *composable* relaxation of linearizability that applies to the anonymous broadcast model?

References

1. Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic Snapshots of Shared Memory. *JACM*, 40(4):873–890, 1993.
2. 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.
3. James Aspnes, Faith Ellen Fich, and Eric Ruppert. Relationships between broadcast and shared memory in reliable anonymous distributed systems. *Distributed Computing*, 18(3):209–219, 2006.
4. Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing Memory Robustly in Message-passing Systems. *J. ACM*, 42(1):124–142, January 1995. doi:10.1145/200836.200869.
5. François Bonnet and Michel Raynal. The Price of Anonymity: Optimal Consensus Despite Asynchrony, Crash, and Anonymity. *ACM Trans. Auton. Adapt. Syst.*, 6(4):23:1–23:28, October 2011. doi:10.1145/2019591.2019592.

6 Zohir Bouzid, Michel Raynal, and Pierre Sutra. Anonymous obstruction-free (n, k)-set agreement with n-k+1 atomic read/write registers. *Distributed Computing*, 31(2):99–117, 2018.

7 Zohir Bouzid, Pierre Sutra, and Corentin Travers. Anonymous Agreement: The Janus Algorithm. In *Principles of Distributed Systems - 15th International Conference, OPODIS 2011, Toulouse, France, December 13-16, 2011. Proceedings*, pages 175–190, 2011.

8 Claire Capdevielle, Colette Johnen, Petr Kuznetsov, and Alessia Milani. On the uncontended complexity of anonymous agreement. *Distributed Computing*, 30(6):459–468, 2017.

9 S. Chaudhuri. More Choices Allow More Faults: Set Consensus Problems in Totally Asynchronous Systems. *Information and Computation*, 105(1):132–158, 1993.

10 Tom Chothia and Konstantinos Chatzikokolakis. A Survey of Anonymous Peer-to-Peer File-Sharing. In *Proceedings of Satellite Workshop of the International Conference on Embedded and Ubiquitous Systems (EUS)*, pages 744–755, 2005.

11 Carole Delporte-Gallet and Hugues Fauconnier. Two Consensus Algorithms with Atomic Registers and Failure Detector Ω. In *Proceedings of the 10th International Conference on Distributed Computing and Networking (ICDCN)*, volume 5408 of *Lecture Notes in Computer Science (LNCS)*, pages 251–262, 2009.

12 Carole Delporte-Gallet, Hugues Fauconnier, Petr Kuznetsov, and Eric Ruppert. On the Space Complexity of Set Agreement? In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*, pages 271–280, 2015.

13 Carole Delporte-Gallet, Hugues Fauconnier, Sergio Rajsbaum, and Nayuta Yanagisawa. A Characterization of t-Resilient Solvable Colorless Tasks in Anonymous Shared-Memory Model. In *SIROCCO2018 (to appear)*, 2018. Technical report: `https://arxiv.org/abs/1712.04393`.

14 Carole Delporte-Gallet, Hugues Fauconnier, Sergio Rajsbaum, and Nayuta Yanagisawa. An anonymous wait-free weak-set object implementation. In *NETYS2018 (to appear)*, 2018.

15 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32(2):374–382, April 1985.

16 Rachid Guerraoui and Eric Ruppert. Anonymous and fault-tolerant shared-memory computing. *Distributed Computing*, 20(3):165–177, 2007.

17 Maurice Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13(1):123–149, 1991.

18 Maurice Herlihy, Dmitry Kozlov, and Sergio Rajsbaum. *Distributed computing through combinatorial topology.* Morgan Kaufmann, 2013.

19 Maurice Herlihy and Sergio Rajsbaum. A classification of wait-free loop agreement tasks. *Theoretical Computer Science*, 291(1):55–77, 2003.

20 Nayuta Yanagisawa. Wait-Free Solvability of Colorless Tasks in Anonymous Shared-Memory Model. *Theory of Computing Systems*, pages pp 1–18, November 2017. `doi:10.1007/s00224-017-9819-0`.

Optimal Rendezvous $\mathcal{L}$-Algorithms for Asynchronous Mobile Robots with External-Lights

Takashi Okumura
Graduate School of Science and Engineering, Hosei University, Tokyo, 184-8485, Japan
takashi.okumura.4e@stu.hosei.ac.jp

Koichi Wada[1]
Faculty of Science and Engineering, Hosei University, Tokyo, 184-8485, Japan
http://ai.k.hosei.ac.jp/staff/wada
wada@hosei.ac.jp

Xavier Défago
School of Computing, Tokyo Institute of Technology, Tokyo, Japan
defago@c.titech.ac.jp

https://orcid.org/0000-0002-2377-205X

Abstract

We study the *Rendezvous* problem for two autonomous mobile robots in asynchronous settings with persistent memory called *light*. It is well known that Rendezvous is impossible in a basic model when robots have no lights, even if the system is semi-synchronous. On the other hand, Rendezvous is possible if robots have lights of various types with a constant number of colors. If robots can observe not only their own lights but also other robots' lights, their lights are called *full-light*. If robots can only observe the state of other robots' lights, the lights are called *external-light*. This paper focuses on robots with external-lights in asynchronous settings and a particular class of algorithms called $\mathcal{L}$-algorithms, where an $\mathcal{L}$-algorithm computes a destination based only on the current colors of observable lights. When considering $\mathcal{L}$-algorithms, Rendezvous can be solved by robots with full-lights and three colors in general asynchronous settings (called ASYNC) and the number of colors is optimal under these assumptions. In contrast, there exist no $\mathcal{L}$-algorithms in ASYNC with external-lights regardless of the number of colors.

In this paper, extending the impossibility result, we show that there exist no $\mathcal{L}$-algorithms in so-called LC-1-Bounded ASYNC with external-lights regardless of the number of colors, where LC-1-Bounded ASYNC is a proper subset of ASYNC and other robots can execute at most one *Look* operation between the *Look* operation of a robot and its subsequent *Compute* operation. We also show that LC-1-Bounded ASYNC is the minimal subclass in which no $\mathcal{L}$-algorithms with external-lights exist. That is, Rendezvous can be solved by $\mathcal{L}$-algorithms using external-lights with a finite number of colors in LC-0-Bounded ASYNC (equivalently LC-atomic ASYNC). Furthermore, we show that the algorithms are optimal in the number of colors they use.

2012 ACM Subject Classification Theory of computation → Distributed algorithms, Computing methodologies → Self-organization

Keywords and phrases Autonomous mobile robots, Rendezvous, Lights, L-algorithms

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.24

Acknowledgements This research was partly supported by JSPS KAKENHI No. 17K00019 and by Japan Science and Technology Agency (JST) SICORP.

[1] contact author

© Takashi Okumura, Koichi Wada, and Xavier Défago;
licensed under Creative Commons License CC-BY
22nd International Conference on Principles of Distributed Systems (OPODIS 2018).
Editors: Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira; Article No. 24; pp. 24:1–24:16

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

1 Introduction

Background and Motivation. The computational issues of autonomous mobile robots have been the object of much research in the field of distributed computing. In particular, a large amount of work has been dedicated to the research of theoretical models of autonomous mobile robots [1, 2, 3, 6, 12, 15, 19, 20]. In the basic common setting, a robot is modeled as a point in a two dimensional plane and its capability is quite weak. We usually assume that robots are *oblivious* (no memory to record past history), *anonymous* and *uniform* (robots have no IDs and run identical algorithms) [8]. Robots operate in *Look*-*Compute*-*Move* (*LCM*) cycles in the model. In the *Look* operation, robots obtain a snapshot of the environment and they execute the same algorithm using the snapshot as input for the *Compute* operation, and move towards the computed destination in the *Move* operation. Repeating these cycles, all robots collectively perform a given task. The weak capabilities of the robots make it challenging for them to accomplish even simple tasks. Therefore, identifying the minimum (weakest) capabilities that the robots need to complete a given task in a given model constitutes a very interesting and important challenge for the theoretical research on autonomous mobile robots.

This paper considers the problem of *Gathering*, which is one of the most fundamental tasks for autonomous mobile robots. Gathering is the process where n mobile robots, initially located at arbitrary positions, meet within finite time at a location, not known a priori. When there are two robots in this setting (i.e., for $n = 2$), this task is called *Rendezvous*. In this paper, we focus on Rendezvous in asynchronous settings and we reveal the relationship among several assumptions.

Since Gathering and Rendezvous are simple but essential problems, they have been intensively studied and a number of possibility and/or impossibility results have been shown under the different assumptions [1, 2, 3, 5, 6, 7, 9, 13, 14, 15, 16, 18, 19]. The solvability of Gathering and Rendezvous depends on the activation schedule and the synchronization level. Usually three basic types of schedulers are identified, namely, the fully synchronous (FSYNC), the semi-synchronous (SSYNC) and the asynchronous (ASYNC) models. In the FSYNC model, there is a common round and in each round all robots are activated simultaneously and *Compute* and *Move* are done instantaneously. The SSYNC model is the same as FSYNC except that at each round only a subset of the robots are activated, with a fairness guarantee that every robot is activated infinitely often in any infinite execution. In the ASYNC scheduler, there are no restrictions about the notion of time. In particular, *Compute* and *Move* and the interval between them can take any (finite) duration, a robot can be seen while moving, and in the interval between an observation and a corresponding move other robots may have possibly moved several times. Gathering and Rendezvous are trivially solvable in FSYNC in the basic model (e.g., without lights) by using an algorithm that moves to the center of gravity. However, these problems can not be solved in SSYNC without any additional assumptions [8].

Das *et al.* [4] extend the classical model with persistent memory, called *lights*, to reveal the relationship between ASYNC and SSYNC and they show that asynchronous robots equipped with lights and a constant number of colors, are strictly more powerful than semi-synchronous robots without lights. In order to solve Rendezvous without any other additional assumptions, robots with lights have been introduced [10, 4, 21]. Table 1 shows previous results including ours to solve Rendezvous by robots with lights, for each scheduler and movement restriction. In the table, *LC*-atomic ASYNC is a subclass of ASYNC, in which we consider from the beginning of each *Look* operation to the end of the corresponding *Compute* operation as an

Table 1 Rendezvous algorithms by robots with lights.

scheduler	movement	full-light	external-light	internal-light	no-light
FSYNC	Non-Rigid	–	–	–	◯ [8]
SSYNC	Non-Rigid	**2**∗(S) [21]	**3**∗(S) [21]	∞∗ [10]	× [8]
	Rigid	–	–	6 [10]	
	Non-Rigid(+δ)	–	–	3 [10]	
LC-atomic ASYNC	Non-Rigid	**2***(S) [17]	?→ **4***(QS),**5***(S)	?	–
	Rigid	–	? → **3***	?	
	Non-Rigid(+δ)	–	?	?	
ASYNC	Non-Rigid	**2**(S) [11],3*(S) [21]	∞∗ [10]	?	–
	Rigid	**2*** [21]	12 [10]	?	
	Non-Rigid(+δ)	–	3 [10]	?	

◯: solvable, ×: unsolvable. ∗: $\mathcal{L}$-algorithm, (S): self-stabilizing,
(QS): quasi-self-stabilizing. – indicates that this part has been solved under weaker conditions or unsolved under stronger ones. A number represents the number of colors used in these algorithms and it is in **boldface** when optimal. ? means that this part has not been solved.

atomic one, that is, no robot can observe between the beginning of each *Look* operation and the end of the next *Compute* on the same robot [17]. Regarding the various kinds of lights, *full-light* means that robots can see their own light as well as that of the other robots, whereas *external-light* and *internal-light* respectively mean that they can see only the lights of the other robots, or only their own light. Regarding the movement restriction, *Rigid* means that the robots can always reach the computed destination during the move operation. *Non-Rigid* means that robots may be stopped before reaching the computed destination but move a minimum distance $\delta > 0$. Non-Rigid(+δ) means it is Non-Rigid and robots know the value δ.

In Table 1, we can see that complete solutions have been obtained for the case of full-lights. However, the cases of external-lights and internal-lights are still insufficiently explored and should be solved.

Our Contribution. In this paper, we are concerned with Rendezvous for robots equipped with external-lights and a particular class of algorithms called $\mathcal{L}$-algorithms. Briefly, an $\mathcal{L}$-algorithm means that each robot (1) always computes a destination on the line connecting the two robots, and (2) using only the observed colors of the lights of the robots.

Algorithms of this class are of interest because they operate also when the coordinate system of a robot is not self-consistent (i.e., it can unpredictably rotate, change its scale or undergo a reflection) [10]. Rendezvous can be solved by an $\mathcal{L}$-algorithm with 3 colors of external-lights in SSYNC [21], but cannot be solved by any $\mathcal{L}$-algorithm with any number of colors of external-lights in ASYNC [10].

In this paper, we reveal the relationship among the number of colors, movement restrictions and initial configurations on $\mathcal{L}$-algorithms with external-lights in asynchronous settings. We introduce subclasses of ASYNC called *LC*-k-Bounded ASYNC ($k \geq 0$), where *LC*-k-Bounded ASYNC is a subclass of ASYNC in which any other robot can execute at most k *Look* operations between the *Look* operation of a robot and its subsequent *Compute* one. When $k = 0$, it is equivalent to *LC*-atomic ASYNC and any *Look* operation and its subsequent *Compute* one can be executed atomically and this interval cannot be observed by any other

robots. We show that Rendezvous cannot be solved by any $\mathcal{L}$-algorithm with any number of colors of external-lights in LC-1-Bounded ASYNC and Rendezvous can be solved by $\mathcal{L}$-algorithm with a finite number of colors of external-lights in LC-0-Bounded (LC-atomic) ASYNC. In fact, we give three $\mathcal{L}$-algorithms with external-lights in LC-atomic ASYNC, such that (1) if we may start from a particular initial configuration with the same color, Rendezvous is solved with 3 colors in Rigid, (2) if we start from any initial configuration with the same color (called *quasi-self-stabilizing*), Rendezvous is solved with 4 colors in Non-Rigid, and (3) if we start from any initial configuration (called *self-stabilizing*), Rendezvous is solved with 5 colors and in Non-Rigid. We also show that the numbers of colors used in the three algorithms are optimal in the sense that no $\mathcal{L}$-algorithm with fewer colors can solve Rendezvous. In order to derive the lower bounds we give several essential properties of $\mathcal{L}$-algorithms.

The remainder of the paper is organized as follows. In Section 2, we define the robot model with lights, the Rendezvous problem, and basic terminology. Section 3 reviews previous results on Rendezvous with lights and the impossibility result of $\mathcal{L}$-algorithms with external-lights is extended. Section 4 shows several properties of $\mathcal{L}$-algorithms for Rendezvous with 3 colors of external-lights and Section 5 shows optimal Rendezvous $\mathcal{L}$-algorithms on Asynchronous robots with external-lights. Section 6 concludes the paper.

2 Preliminaries

2.1 Robot Model

We consider a set of n anonymous mobile robots $\mathcal{R} = \{r_1, \dots, r_n\}$ located in $\mathbb{R}^2$. Each robot r_i has a persistent state $\ell(r_i)$ called *light* which may be taken from a finite set of colors L.

We denote by $\ell(r_i, t)$ the color that the light of robot r_i has at time t and $p(r_i, t) \in \mathbb{R}^2$ the position occupied by r_i at time t represented in some global coordinate system. Given two points p and $q \in \mathbb{R}^2$, $dis(p, q)$ denotes the distance between p and q.

Each robot r_i has its own coordinate system where r_i is located at its origin at any time. These coordinate systems do not necessarily agree with those of other robots. It means that there is no common knowledge of unit of distance, directions of its coordinates, or clockwise orientation (*chirality*).

At any point of time, a robot can be active or inactive. When a robot r_i is activated, it executes *Look*-*Compute*-*Move* operations:

- *Look*: The robot r_i activates its sensors to obtain a snapshot which consists of a pair of light and position for every robot with respect to the coordinate system of r_i. Since the result of this operation is a snapshot of the positions of all robots, the robot does not notice the movement, even if it sees other moving robots. We assume that robots can observe all other robots (unlimited visibility).
- *Compute*: The robot r_i executes its algorithm using the snapshot and the color of its own light (if allowed by the model) and returns a destination point des_i expressed in its coordinate system and a light $\ell_i \in L$ to which its own color is set.
- *Move*: The robot r_i moves to the computed destination des_i. A robot r is said to *collide* with robot s at time t if $p(r, t) = p(s, t)$ and at time t r is performing *Move*. The collision is *accidental* if r's destination is not $p(r, t)$. Since robots are seen as points, we assume that accidental collisions are immaterial. A moving robot, upon causing an accidental collision, proceeds in its movement without changes, in a "hit-and-run" fashion [8]. The robot may be stopped by an adversary before reaching the computed destination. If stopped before reaching its destination, a robot moves at least a minimum distance $\delta > 0$.

Note that without this assumption an adversary could make it impossible for any robot to ever reach its destination. If the distance to the destination is at most δ, the robot can reach it. In this case, the movement is called *Non-Rigid*. Otherwise, it is called *Rigid*. If the movement is Non-Rigid and robots know the value of δ, it is called *Non-Rigid(+δ)*.

A scheduler decides which subset of robots is activated for every configuration. The schedulers we consider are asynchronous and semi-synchronous and it is assumed that schedulers are *fair*, each robot is activated infinitely often.

- **ASYNC:** The asynchronous (ASYNC) scheduler, activates the robots independently, and the duration of each *Compute*, *Move* and between successive activities is finite and unpredictable. As a result, robots can be seen while moving and the snapshot and its actual configuration are not the same and so its computation may be done with the old configuration.
- **SSYNC:** The semi-synchronous(SSYNC) scheduler activates a subset of all robots synchronously and their *Look*-*Compute*-*Move* cycles are performed at the same time. We can assume that activated robots at the same time obtain the same snapshot and their *Compute* and *Move* are executed instantaneously. In SSYNC, we can assume that each activation defines discrete time called *round* and *Look*-*Compute*-*Move* is performed instantaneously in one round.

As a special case of SSYNC, if all robots are activated in each round, the scheduler is called full-synchronous (FSYNC).

In this paper, we are concerned with ASYNC and we assume the followings; In a *Look* operation, a snapshot of the environment at time t_L is taken and we say that the *Look operation is performed at time* t_L. Each *Compute* operation of r_i is assumed to be done at time t_C and the color of its light $\ell_i(t)$ and its pending destination des_i are both set to the computed values for any time greater than t_C[2]. In a *Move* operation, when the movement begins at time t_B and ends at t_E, we say that it is performed during interval $[t_B, t_E]$, and the beginning (resp. ending) of the movement is denoted by $Move_{BEGIN}$ (resp. $Move_{END}$) occurring at time t_B (resp. t_E). In the following, *Compute*, $Move_{BEGIN}$ and $Move_{END}$ are abbreviated as *Comp*, M_B and M_E, respectively. When a cycle has no actual movement (i.e., robots only change color and their destinations are the current positions), we can equivalently assume that the *Move* operation in this cycle is omitted, since we can consider the *Move* operation to be performed just before the next *Look* operation.

Without loss of generality, we assume the set of time instants at which the robots start executions of *Look*, *Comp*, M_B and M_E is $\mathbb{N}$.

We also consider the following restricted classes of ASYNC. Let k be a non-negative integer. Let a robot r execute a cycle. If any other robot can execute at most k *Look* operations between the *Look* operation of r and its subsequent *Compute* in that cycle, the model is said to be *LC-k-Bounded*. If $k = 0$, it is said to be *LC-atomic*. Thus we can assume that in the *LC*-atomic ASYNC model, *Look* and *Comp* operations in every cycle are performed simultaneously (or atomically), say at time t_{LC}, and we say that the *LC*-operation is performed at time t_{LC}.

Similarly, if no other robot can execute at most *Look* operations between the operation M_B of r and its corresponding M_E, the model is said to be *Move-k-Bounded*. If $k = 0$, it is said to be *Move-atomic*. In this case *Move* operations in all cycles can be considered to

[2] Note that if some robot performs a *Look* operation at time t_C, then it observes the former color and if it does at time $t_C + \epsilon (\forall \epsilon > 0)$, then it observes the newly computed color.

be performed instantaneously and at time t_M. In *Move*-atomic ASYNC, when a robot r observes another robot r' performing a *Move* operation at time t_M, r observes the snapshot after the moving of r'.

Since each operation occurs at integer times, when LC-operation is performed at time t in LC-atomic ASYNC, we can assume that the snapshot at t is obtained at t and the computation completes at $t+1$. Also when *Move*-operation begins (M_B occurs) at time t in *Move*-atomic ASYNC, M_E can be assumed to occur at time $t+1$. Thus, if a robot r observes another robot r' performing a *Move* operation at time t_M, then r observes the snapshot before the moving of r' until and at time t, and the snapshot after the moving of r' from t_M+1.

In our settings, robots have persistent lights and can change their colors instantly at each *Compute* operation. We consider the following three robot models according to the visibility of lights.

- *full-light*, a robot can observe the lights of other robots as well as its own, and it can also change the color of its own light.
- *external-light*, a robot can observe the light of other robots but not its own. It can however change the color of its own light in a "write-only" manner.
- *internal-light*, a robot can observe and change the color of its own light, but cannot observe the lights of other robots.

2.2 Rendezvous and $\mathcal{L}$-Algorithms

An n-*Gathering* problem is defined as follows: given $n(\geq 2)$ robots initially placed at arbitrary positions in $\mathbb{R}^2$, they congregate in finite time at a single location which is not predefined. In the following, we consider the case where $n=2$ and the 2-Gathering problem is called *Rendezvous*.

When we consider algorithms on robots with lights, we exclude algorithms that solve Rendezvous only starting from initial settings in which robots have different colors of lights. That is, we consider Rendezvous algorithms that can solve Rendezvous even from initial settings in which all robots have the same color. An algorithm solving Rendezvous is said to be *quasi-self-stabilizing* if it assumes that both robots always start with the same initial color chosen arbitrarily, and it is *self-stabilizing* if the robots can start from arbitrary colors.

A particular class of algorithms, denoted by $\mathcal{L}$, requires that robots only compute a destination point of the form $(1-\lambda)\cdot \text{me.position} + \lambda \cdot \text{other.position}$ for some $\lambda \in \mathbb{R}$, obtained as a function having only the colors as input (i.e., color of the other robot in the external-light) [21]. We call an algorithm in this class an $\mathcal{L}$-algorithm.

3 Previous Results for Rendezvous

Rendezvous is trivially solvable in FSYNC but is not in SSYNC in general.

▶ **Theorem 1.** [8] *Rendezvous is deterministically unsolvable in SSYNC even if chirality is assumed.*

If robots have a constant number of colors in their lights, Rendezvous can be solved as shown in the following theorems (or Table 1).

▶ **Theorem 2.** *Rendezvous is solved by self-stabilizing $\mathcal{L}$-algorithms under the following assumptions;*

1. *full-light with* 2 *colors, Non-Rigid and LC-atomic ASYNC* [17]*,*
2. *full-light with* 3 *colors, Non-Rigid and ASYNC* [21]*,*
3. *external-light with* 3 *colors, Non-Rigid and SSYNC* [10]*.*

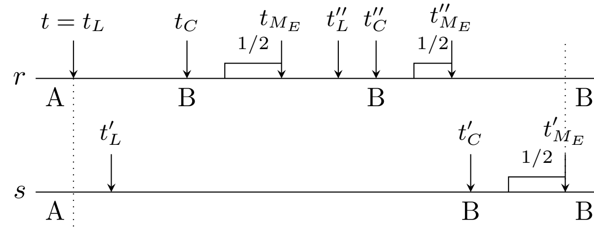


Figure 1 Move-atomic and LC-1-Bounded ASYNC schedule Rendezvous never succeeds.

▶ **Theorem 3.** [11] *Rendezvous is solved by a self-stabilizing non-$\mathcal{L}$-algorithm in full-light with* 2 *colors, Non-Rigid and ASYNC.*

▶ **Theorem 4.** [10] *Rendezvous is solved by non-quasi-self-stabilizing non-$\mathcal{L}$-algorithms under the following assumptions;*

1. *external-light with* 3 *colors, Non-Rigid($+\delta$) and ASYNC,*
2. *external-light with* 12 *colors, Rigid and ASYNC,*
3. *internal-light with* 3 *colors, Non-Rigid($+\delta$) and SSYNC,*
4. *internal-light with* 6 *colors, Rigid and SSYNC.*

Impossibility of Rendezvous $\mathcal{L}$-algorithms is stated as follows.

▶ **Theorem 5.**

1. *In ASYNC and Rigid, Rendezvous is not solvable by any quasi-self-stabilizing $\mathcal{L}$-algorithm with full-light of* 2 *colors* [21]*.*
2. *In ASYNC and Non-Rigid, Rendezvous is not solvable by any $\mathcal{L}$-algorithm with full-light of* 2 *colors* [21]*.*
3. *In Move-atomic but non-LC-atomic ASYNC and Rigid, Rendezvous is not solvable by any $\mathcal{L}$-algorithm with external-light of any number of colors* [10]*.*
4. *In SSYNC and Rigid, Rendezvous is not solvable by any $\mathcal{L}$-algorithm with internal-light of any number of colors* [10]*.*

Theorem 5 point **3** can be extended as follows;

▶ **Theorem 6.** *In Move-atomic and LC-*1*-Bounded ASYNC, and Rigid, Rendezvous is not solvable by any $\mathcal{L}$-algorithm with external-light of any number of colors.*

Proof. For each robot, the destination point and the next color are a function of the color of the other robot only. Assume that both robots start in the same color (say, A) and perform their execution synchronously. Consider a time t when both robots compute the midpoint m as a result of looking each other color A. Only robot r is let begin the cycle and perform *Look* operation, *Compute* one and M_E one at $t_L = t$, t_C and t_{M_E}, respectively. Robot r computes the midpoint m and changes its color to say, B at t_C. Robot s is let perform *Look* operation at time t'_L ($t_L < t'_L < t_C$), compute the midpoint m and change its color to B at time t'_C ($t_{M_E} < t'_C$), and move to m at time t'_{M_E} (Figure 1). Robot r is let end the next cycle before t'_C and perform *Look* operation, *Compute* one and M_E one at t''_L, t''_C and t'_{M_E}, respectively, where $t''_{M_E} < t'_C$. Since r keeps seeing s set to A in this cycle, r computes the new midpoint m' and changes its color to B. Then, both robots have the same color and does not attain Rendezvous at the time t'_{M_E}. By repeating the pattern, the robots never attain Rendezvous. Also this pattern satisfies *Move*-atomic and *LC*-1-Bounded ASYNC, and Rigid. ◀

Algorithm 1 SS-Rendezvous-with-3-colors (scheduler, movement, initial-color)[10].

Parameters: scheduler, movement-restriction, initial-color
Assumptions: external-light, three colors (A, B and C)

1: **case** other.light **of**
2: A:
3: $me.light \leftarrow B$
4: $me.des \leftarrow$ the midpoint of $me.position$ and $other.position$
5: B:
6: $me.light \leftarrow C$
7: C:
8: $me.light \leftarrow A$
9: $me.des \leftarrow other.position$
10: **endcase**

In the following sections, we consider $\mathcal{L}$-algorithms to solve Rendezvous on robots with external-lights and clarify the relationship among synchrony, the number of colors, movement restriction, and initial configurations.

4 Rendezvous $\mathcal{L}$-Algorithms for Robots with Three Colors of External Lights

In what follows, two robots are denoted as r and s. Let t_0 be the starting time of the algorithm. Given a robot *robot*, an operation $op(\in \{Look, Comp, LC, M_B, M_E\})$, and a time t, $t^+(robot, op)$ denotes the time *robot* performs the first *op* after t (inclusive) if there exists such operation, and $t^-(robot, op)$ denotes the time *robot* performs the first *op* before t (inclusive) if there exists such operation. If t is the time the algorithm terminates, $t^+(robot, op)$ is not defined for any *op*. When *robot* does not perform *op* before t and $t^-(robot, op)$ does not exist, $t^-(robot, op)$ is defined to be t_0.

A time t_c is called a *cycle start time* (*cs-time*, for short), if the next performed operations of both r and s after t_c are both *Look*, or otherwise, the robots performing the operations neither change their colors of lights nor move. In the latter case, we can consider that these operations can be performed before t_c and the subsequent *Look* operation can be performed as the first operation after t_c.

In [10], a Rendezvous algorithm is shown in SSYNC and Non-Rigid with external-light of three colors (Algorithm 1).

▶ **Theorem 7.** [10] *Rendezvous is solved by SS-Rendezvous-with-3-colors(SSYNC, Non-Rigid, any). It is a self-stabilizing $\mathcal{L}$-algorithm.*

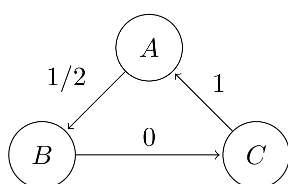
We will show that Algorithm 1 does not work in even LC-atomic and *Move*-atomic ASYNC and Rigid, starting from the initial color A. In fact, in the next section, more generally we will show that there exist no $\mathcal{L}$-algorithms to solve Rendezvous in LC-atomic and *Move*-atomic ASYNC and Non-Rigid with three colors of external-lights. We also show that there exist no quasi-self-stabilizing $\mathcal{L}$-algorithms to solve Rendezvous if we change the assumption of Non-Rigid to Rigid. On the other hand, we show that there exists a non-quasi-self-stabilizing $\mathcal{L}$-algorithm to solve Rendezvous in LC-atomic ASYNC and Rigid with three colors of external-lights (Algorithm 2).

▶ **Theorem 8.** *Rendezvous is solved by NonQSS-Rendezvous-with-3-colors (LC-atomic ASYNC, Rigid, A). It is a non-quasi-self-stabilizing $\mathcal{L}$-algorithm.*

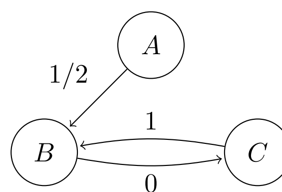
Algorithm 2 NonQSS-Rendezvous-with-3-colors (scheduler, movement, initial-color).

Parameters: scheduler, movement-restriction, initial-color
Assumptions: external-light, three colors (A, B and C)

1: **case** other.light **of**
2: A:
3: $me.light \leftarrow B$
4: $me.des \leftarrow$ the midpoint of $me.position$ and $other.position$
5: B:
6: $me.light \leftarrow C$
7: C:
8: $me.light \leftarrow B$
9: $me.des \leftarrow other.position$
10: **endcase**



(a) Algorithm 1



(b) Algorithm 2

Figure 2 Graph representations for Algorithms 1 (a) and 2 (b).

In the following, we derive lower bounds on the number of colors of external-lights. In order to do so, we introduce some notation for $\mathcal{L}$-algorithms and show their properties.

In $\mathcal{L}$-algorithms, the next color and destination (denoted as λ) is determined only by the current color observed by the robot. Thus an $\mathcal{L}$-algorithm is represented by an edge-labeled directed graph $G_{\mathcal{L}} = (V_{\mathcal{L}}, E_{\mathcal{L}}, \ell_{\mathcal{L}})$, where $V_{\mathcal{L}}$ is a set of colors used in the algorithm, $E_{\mathcal{L}}$ is a set of transitions from current colors observed by the robots to the next colors computed by the robots, and $\ell_{\mathcal{L}}$ is an edge-labeled function from $E_{\mathcal{L}}$ to $\mathbb{R}$. Edge $e = (c_1, c_2) \in E_{\mathcal{L}}$ and $\ell_{\mathcal{L}}(e) = \lambda$ mean that when a robot observes color c_1 of the other robot, it changes its color to c_2 and moves to the point decided by the value λ.[3] Also the out-degree of each node must be one, since we consider deterministic $\mathcal{L}$-algorithms. Thus, when the number of nodes in $G_{\mathcal{L}}$ is k, $G_{\mathcal{L}}$ has k edges. For example, Algorithms 1 and 2 are represented by the following directed graphs $G_{\mathcal{L}1}$ and $G_{\mathcal{L}2}$, respectively.

$G_{\mathcal{L}1} = (V_{\mathcal{L}1}, E_{\mathcal{L}1}, \ell_{\mathcal{L}1})$, where $V_{\mathcal{L}1} = \{A, B, C\}$, $E_{\mathcal{L}1} = \{(A, B), (B, C), (C, A)\}$ and $\ell_{\mathcal{L}1}((A, B)) = 1/2$, $\ell_{\mathcal{L}1}((B, C)) = 0$ and $\ell_{\mathcal{L}1}((C, A)) = 1$ (Figure 2(a)).

$G_{\mathcal{L}2} = (V_{\mathcal{L}2}, E_{\mathcal{L}2}, \ell_{\mathcal{L}2})$, where $V_{\mathcal{L}2} = \{A, B, C\}$, $E_{\mathcal{L}2} = \{(A, B), (B, C), (C, B)\}$ and $\ell_{\mathcal{L}2}((A, B)) = 1/2$, $\ell_{\mathcal{L}1}((B, C)) = 0$ and $\ell_{\mathcal{L}2}((C, B)) = 1$ (Figure 2(b)).

In what follows, we identify an $\mathcal{L}$-algorithm with its graph representation and $e = (c_1, c_2) \in E_{\mathcal{L}}$ and $\ell_{\mathcal{L}}(e) = \lambda$ are denoted as $c_1 \xrightarrow{\lambda} c_2$.

▶ **Lemma 9.** *Let $A_{\mathcal{L}}$ be an $\mathcal{L}$-algorithm solving Rendezvous in SSYNC and Rigid with external-light. If $A_{\mathcal{L}}$ starts from an initial setting such that both robots have the same color, then $A_{\mathcal{L}}$ has the following properties.*

[3] Note that $G_{\mathcal{L}}$ is not a state-transition graph.

1. *There is a color* X *such that* $A_{\mathcal{L}}$ *must have an edge* $X \stackrel{1/2}{\rightarrow} Y$*.*
2. *There is a color* X *such that* $A_{\mathcal{L}}$ *must have an edge* $X \stackrel{1}{\rightarrow} Y$*.*
3. *There is a color* X *such that* $A_{\mathcal{L}}$ *must have an edge* $X \stackrel{0}{\rightarrow} Y$*.*

Lemma 9 implies that any $\mathcal{L}$-algorithm must contain three different edges beginning with different colors.

▶ **Theorem 10.** [4] *In any Rendezvous $\mathcal{L}$-algorithm with external-light, robots must have three colors in SSYNC and Rigid.*

This theorem implies that Algorithm 1 has the optimal number of colors of external-lights in SSYNC. Note that it is self-stabilizing and works in Non-Rigid. On the other hand, if we assume Rigid movement, we can show the $\mathcal{L}$-algorithm with three colors to solve Rendezvous in LC-atomic ASYNC, which is however not quasi-self-stabilizing. In the next section, we will show a quasi-self-stabilizing $\mathcal{L}$-algorithm with four colors and a self-stabilizing $\mathcal{L}$-algorithm with five colors to solve Rendezvous in LC-atomic ASYNC and Non-Rigid. We will also show that the number of colors used in each algorithm is optimal.

5 Optimal Rendezvous $\mathcal{L}$-Algorithms for LC-atomic ASYNC Robots with External Lights

5.1 Lower Bounds

In this subsection, we first show that there exist no Rendezvous $\mathcal{L}$-algorithms with external light of 3 colors in LC-atomic and $Move$-atomic ASYNC in Non-Rigid.

If there exists such an $\mathcal{L}$-algorithm, the algorithm must be an edge-labeled directed graph $G_{\mathcal{L}} = (V_{\mathcal{L}}, E_{\mathcal{L}}, \ell_{\mathcal{L}})$ such that $V_{\mathcal{L}} = \{A, B, C\}$(three colors) and $\ell_{\mathcal{L}}(E_{\mathcal{L}}) = \{0, 1/2, 1\}$ (by Lemma9) and one of the following edge sets:

1. $E_{\mathcal{L}}$ contains a self-loop edge, say (A, A), and does not contain both directed edges,
2. $E_{\mathcal{L}}$ contains both directed edges, say (B, C) and (C, B), or
3. $E_{\mathcal{L}} = \{(A, B), (B, C), (C, A)\}$.

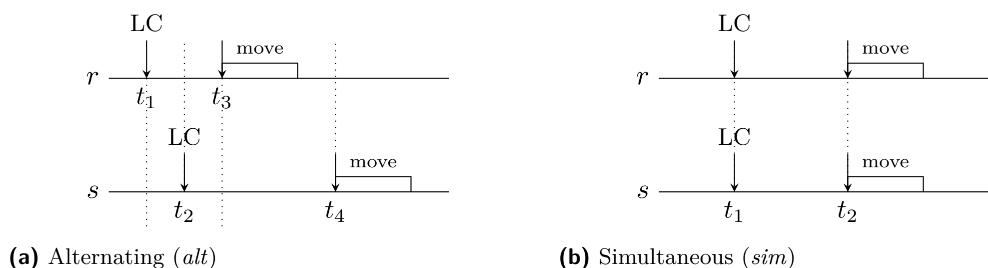
For Case 1. If the algorithm does not contain both directed edges, it can be verified that no algorithm can solve Rendezvous in SSYNC and Rigid. That is, if the algorithm starts with a color consisting of a self-loop edge, then it cannot solve Rendezvous since it cannot use more than one color. If the algorithm starts with a color not consisting of a self-loop edge, the color of both robots can be changed into the color with the self-loop edge without attaining Rendezvous. Thus, the algorithm also fails to Rendezvous in this case.

For Case 2. If algorithms do not contain self-loop edges, their graphs are the same as that of Algorithm 2. But it can be verified that Algorithm 2 fails to solve Rendezvous in SSYNC, Rigid and starting from color B or C, or SSYNC, Non-Rigid and starting from any color. It is easily verified that other algorithms with different edge-labeled functions fail to solve Rendezvous in SSYNC and Rigid starting from any color. If algorithms contain self-loop edges (both directed edges and a self-loop edge), since they can use only less than three colors even if starting from any color, they never solve Rendezvous in SSYNC and Rigid.

In Case 3, there are essentially two algorithms.

(a) $\ell_{\mathcal{L}}((A, B)) = 1/2$, $\ell_{\mathcal{L}}((B, C)) = 0$, and $\ell_{\mathcal{L}}((C, A)) = 1$ (denoted as Alg-(a)),
(b) $\ell_{\mathcal{L}}((A, B)) = 1/2$, $\ell_{\mathcal{L}}((B, C)) = 1$, and $\ell_{\mathcal{L}}((C, A)) = 0$ (denoted as Alg-(b)).

[4] This result is stated in [10, 21] but is not proved yet.



(a) Alternating (*alt*) **(b)** Simultaneous (*sim*)

Figure 3 Special schedules *alt* and *sim*.

Note that Alg-(a) is Algorithm 1.

We introduce special schedules to analyze $\mathcal{L}$-algorithms solving Rendezvous in LC-atomic ASYNC, with which we show that these algorithms do not work well.

Let $([\alpha_1, \beta_1], [\alpha_2, \beta_2], \ldots)$ be a sequence of operations that robots r and s perform, where r and s perform α_i and β_i at time t_i $(1 \leq i)$, respectively, and α_i and β_i are taken from LC-operation (denoted as LC), $Move$-operations, M_B, M_E or M (if $Move$-atomic) (denoted as M), and a "*no-op*" operation (denoted as $-$). For example, $([LC, -], [-, LC], [M, -], [-, M])$ denotes that r performs LC and M at times t_1 and t_3 and s performs LC and M at times t_2 and t_4, which is in LC-atomic $Move$-atomic ASYNC. Similarly, $([LC, LC], [M, M])$ denotes that r and s perform LC at time t_1 and perform M at time t_2, which is in FSYNC. The former is called alternating schedule and denoted as *alt* and the latter is called simultaneous schedule and denoted as *sim* (Figure 3).

Assume that r and s have colors c_r and c_s at some time t and let $d_t = dis(p(r,t), p(s,t))$. Let $(c_r, c_s; d_t)$ denote a configuration of a pair of colors of robots and its distance at t. When a configuration $(c_r, c_s; d_t)$ is changed into another one $(c'_r, c'_s; d_{t'})$ by performing an algorithm alg with a schedule sch, we denote $(c_r, c_s; d_t) \stackrel{sch}{\rightarrow} (c'_r, c'_s; d_{t'})_{alg}$, where t' is the time after which the robots have performed alg with the schedule sch. The suffix alg is usually omitted when the algorithm is apparent from the context.

We show that Alg-(a) and Alg-(b) cannot work from any initial configuration of the same color by using the schedules *alt* and *sim*.

▶ **Lemma 11.** *Alg-(a) cannot solve Rendezvous in LC-atomic and Move-atomic ASYNC and Rigid.*

▶ **Lemma 12.** *Alg-(b) cannot solve Rendezvous in LC-atomic and Move-atomic ASYNC and Rigid.*

▶ **Theorem 13.** *There exist no $\mathcal{L}$-algorithms of Rendezvous with external light of 3 colors in LC-atomic and Move-atomic ASYNC and Non-Rigid. Furthermore, there exist no quasi-self-stabilizing $\mathcal{L}$-algorithms of Rendezvous with external light of 3 colors in LC-atomic and Move-atomic ASYNC and Rigid.*

In an argument similar to the one above, we show that there exist no self-stabilizing $\mathcal{L}$-algorithms of Rendezvous with external-light of 4 colors in LC-atomic and $Move$-atomic ASYNC and Rigid.

If there exists such an $\mathcal{L}$-algorithm, the algorithm must be an edge-labeled directed graph $G_\mathcal{L} = (V_\mathcal{L}, E_\mathcal{L}, \ell_\mathcal{L})$ such that $V_\mathcal{L} = \{A, B, C, D\}$(four colors) and $\ell_\mathcal{L}(E_\mathcal{L}) \supseteq \{0, 1/2, 1\}$ (by Lemma 9). If the number of strongly connected components for $G_\mathcal{L}$ is at least two, then there exists an initial configuration of both robots with a same color, from which an algorithm

Algorithm 3 QSS-Rendezvous-with-4-colors (LC-atomic ASYNC, Non-Rigid, initial-color).

Parameters: scheduler, movement-restriction, initial-color
Assumptions: external-light, four colors (A, B, C and D)
1: **case** other.light **of**
2: A:
3: $me.light \leftarrow B$
4: $me.des \leftarrow$ the midpoint of $me.position$ and $other.position$
5: B:
6: $me.light \leftarrow C$
7: C:
8: $me.light \leftarrow D$
9: $me.des \leftarrow other.position$
10: D:
11: $me.light \leftarrow A$
12: **endcase**

cannot use four colors, it cannot solve Rendezvous by Theorem 13. Then the remaining case is that these graphs have one strongly connected component (one cycle) and have one of the following edge sets:

(1) $\ell_{\mathcal{L}}((A,B)) = 1/2$, $\ell_{\mathcal{L}}((B,C)) = 0$, $\ell_{\mathcal{L}}((C,D)) = 1$, and $\ell_{\mathcal{L}}((D,A)) = \lambda$(denoted as Alg-(1)),
(2) $\ell_{\mathcal{L}}((A,B)) = 1/2$, $\ell_{\mathcal{L}}((B,C)) = 1$, $\ell_{\mathcal{L}}((C,D)) = 0$, and $\ell_{\mathcal{L}}((D,A)) = \lambda$(denoted as Alg-(2)),
(3) $\ell_{\mathcal{L}}((A,B)) = 1/2$, $\ell_{\mathcal{L}}((B,C)) = 0$, $\ell_{\mathcal{L}}((C,D)) = \lambda$, and $\ell_{\mathcal{L}}((D,A)) = 1(\lambda \neq 1)$(denoted as Alg-(3)),
(4) $\ell_{\mathcal{L}}((A,B)) = 1/2$, $\ell_{\mathcal{L}}((B,C)) = 1$, $\ell_{\mathcal{L}}((C,D)) = \lambda$, and $\ell_{\mathcal{L}}((D,A)) = 0(\lambda \neq 0)$(denoted as Alg-(4)),
(5) $\ell_{\mathcal{L}}((A,B)) = 1/2$, $\ell_{\mathcal{L}}((B,C)) = \lambda$, $\ell_{\mathcal{L}}((C,D)) = 0$, and $\ell_{\mathcal{L}}((D,A)) = 1(\lambda \neq 1)$(denoted as Alg-(5)),
(6) $\ell_{\mathcal{L}}((A,B)) = 1/2$, $\ell_{\mathcal{L}}((B,C)) = \lambda$, $\ell_{\mathcal{L}}((C,D)) = 1$, and $\ell_{\mathcal{L}}((D,A)) = 0(\lambda \neq 0)$(denoted as Alg-(6)).

▶ **Lemma 14.** *Alg-(1)-Alg-(6) cannot solve Rendezvous in LC-atomic and Move-atomic ASYNC and Rigid from some initial configuration.*

▶ **Theorem 15.** *There exist no self-stabilizing $\mathcal{L}$-algorithms of Rendezvous with external-light of* 4 *colors in LC-atomic and Move-atomic ASYNC and Rigid.*

5.2 Optimal $\mathcal{L}$-algorithms

In this subsection, we show two optimal $\mathcal{L}$-algorithms of Rendezvous, one is quasi-self-stabilizing with 4 colors (Algorithm 3) and the other is self-stabilizing with 5 colors (Algorithm 4).

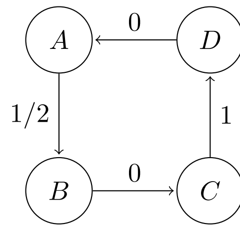
Algorithm 3 (QSS-Rendezvous-with-4-colors (LC-atomic ASYNC, Non-Rigid, initial-light)) satisfies the following lemmas. Let t_c be a cs-time of Algorithm 3.

The correctness proof proceeds as follows;

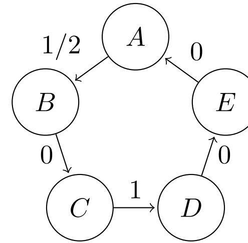
1. First we prove that Algorithm 3 is quasi-self-stabilizing. Algorithm 3 does not work from the initial configuration $\{\ell(r,t_0), \ell(s,t_0)\} = \{A, C\}$ or $\{\ell(r,t_0), \ell(s,t_0)\} = \{B, D\}$ (Lemma 14). However, we show that these configurations can not be reached from the initial configuration that r and s have a same color as follows. Assume that robots r

Algorithm 4 SS-Rendezvous-with-5-colors (LC-atomic ASYNC, Non-Rigid, initial-color).

Parameters: scheduler, movement-restriction, Initial-color)
Assumptions: external-light, five colors (A, B, C, D and E)
1: **case** other.light **of**
2: A:
3: $me.light \leftarrow B$
4: $me.des \leftarrow$ the midpoint of $me.position$ and $other.position$
5: B:
6: $me.light \leftarrow C$
7: C:
8: $me.light \leftarrow D$
9: $me.des \leftarrow other.position$
10: D:
11: $me.light \leftarrow E$
12: E:
13: $me.light \leftarrow A$
14: **endcase**



(a) Algorithm 3



(b) Algorithm 4

Figure 4 Graph representations for Algorithms 3 (a) and 4 (b).

and s start with a same color. If all LC-operations of r and s are performed at the same times, Rendezvous succeeds preserving that $\ell(r,t) = \ell(s,t)$ for any cs-time t (Lemma 16). Otherwise, there are different times at which LC-operations of r and s are performed and let t_r and t_s be the first times r and s are performed LC-operations, respectively ($t_r \neq t_s$). Then we show that there are colors α and β such that $\{\ell(r,t^*), \ell(s,t^*)\} = \{\alpha, \beta\}$ and $\alpha \to \beta$ for any cs-time t^* after the time $max(t_r, t_s)$ (Lemma 17). When robots r and s start with $\{\ell(r,t_c), \ell(s,t_c)\} = \{\alpha, \beta\}$ such that $\alpha \to \beta$, this relation of colors is preserved for any cs-time after t_c (Lemma 18).

2. Next we show that if robots r and s start with colors α and β such that $\alpha \to \beta$, Algorithm 3 attains Rendezvous. If $\{\alpha, \beta\} = \{B, C\}$, Rendezvous succeeds, or there is a cs-time $t^*(\geq t_c)$ such that the distance between r and s at t^* decreases at least δ from the distance at t_c and $\{\ell(r,t^*), \ell(s,t^*)\} = \{C, D\}$ or $\{\ell(r,t^*), \ell(s,t^*)\} = \{D, A\}$ (Lemma 22). If $\{\alpha, \beta\} = \{A, B\}, \{C, D\}$, or $\{D, A\}$, then there is a cs-time $t^*(\geq t_c)$ such that the distance between r and s at t^* is less than or equal to the distance at t_c and $\{\ell(r,t^*), \ell(s,t^*)\} = \{B, C\}$ (Lemma 23).

▶ **Lemma 16.** *Let $\ell(r,t_c) = \ell(s,t_c)$. Assume that all LC-operations of r and s are performed at the same times, and let $t_i (i = 1, 2, 3, \ldots)$ be the times r and s perform LC-operations simultaneously. Then $\ell(r,t_i) = \ell(s,t_i)$ for any time $t_i (i = 1, 2, 3, \ldots)$ and there is a cs-time $t^*(\geq t_c)$ such that $dis(p(r,t^*), p(s,t^*)) = 0$.*

▶ **Lemma 17.** *Let $\ell(r,t_c) = \ell(s,t_c)$. Assume that there are different times at which LC-operations of r and s are performed and let t_r and t_s be the first times r and s are performed LC-operations, respectively ($t_r \neq t_s$). Then there are colors α^* and β^* such that $\{\ell(r,t^*), \ell(s,t^*)\} = \{\alpha^*, \beta^*\}$ and $\alpha^* \to \beta^*$ for the first cs-time t^* after $max(t_r,t_s)$.*

The following lemma can be proved similar to Lemmas 16-17.

▶ **Lemma 18.** *Assume that Algorithm 3 starts with $\{\ell(r,t_c), \ell(s,t_c)\} = \{\alpha, \beta\}$ such that $\alpha \to \beta$. Let t^* be the first cs-time after t_c. Then there are colors α^* and β^* such that $\{\ell(r,t^*), \ell(s,t^*)\} = \{\alpha^*, \beta^*\}$ and $\alpha^* \to \beta^*$.*

▶ **Lemma 19.** *Let $\ell(r,t_c) = \ell(s,t_c)$. If Algorithm 3 is performed starting from t_c, there does not exist any cs-time $t^*(\geq t_c)$ such that $\{\ell(r,t^*), \ell(s,t^*)\} = \{A,C\}$ or $\{\ell(r,t^*), \ell(s,t^*)\} = \{B,D\}$.*

Proof. Configuration of $\{\ell(r,t^*), \ell(s,t^*)\} = \{A,C\}$ or $\{\ell(r,t^*), \ell(s,t^*)\} = \{B,D\}$ at any cs-time t^* cannot be reached from any initial configuration that r and s have a same color by Lemmas 16-18. ◀

▶ **Lemma 20.** *If $dis(p(r,t_c), p(s,t_c)) = 0$ and Algorithm 3 is performed starting from t_c, $dis(p(r,t), p(s,t)) = 0$ for any $t \geq t_c$.*

Proof. Since $dis(p(r,t_c), p(s,t_c)) = 0$, any move operation becomes no move (stay). ◀

▶ **Lemma 21.** *Let $\alpha = B$ and $\beta = C$ or $\alpha = D$ and $\beta = A$. If $\ell(r,t_c) = \alpha$ and $\ell(s,t_c) = \beta$ in Algorithm 3, then $\ell(s,t) = \beta$ and $p(s,t) = p(s,t_c)$ for any $t(t_c \leq t \leq t_1 = t_c^+(r, LC))$.*

Proof. When s with color β observes r with color α at $t(t_c \leq t \leq t_1)$, s does not change its color at time t and stays at position $p(s,t_c)$. ◀

▶ **Lemma 22.** *If Algorithm 3 starts with $\{\ell(r,t_c), \ell(s,t_c)\} = \{B,C\}$, for any schedule of two robots after t_c, there is a cs-time $t^*(\geq t_c)$ such that $dis(p(r,t^*), p(s,t^*)) = 0$, or $dis(p(r,t^*), p(s,t^*)) \leq dis(p(r,t_c), p(s,t_c)) - \delta$ and $\{\ell(r,t^*), \ell(s,t^*)\} = \{C,D\}$ or $\{\ell(r,t^*),\ \ell(s,t^*)\} = \{D,A\}$.*

▶ **Lemma 23.** *If Algorithm 3 starts with $\{\ell(r,t_c), \ell(s,t_c)\} = \{\alpha, \beta\}$ such that $\alpha \to \beta$, for any schedule of two robots after t_c, there is a cs-time $t^*(\geq t_c)$ such that $\{\ell(r,t^*), \ell(s,t_c)\} = \{B,C\}$ and $dis(p(r,t^*), p(s,t^*)) \leq dis(p(r,t_c), p(s,t_c))$.*

Lemmas 16-23 follow the next theorem.

▶ **Theorem 24.** *Rendezvous is solved by QSS-Rendezvous-with-4-colors(LC-atomic ASYNC, Non-Rigid, any) with $\ell(r,t_0) = \ell(s,t_0)$. It is a quasi-self-stabilizing $\mathcal{L}$-algorithm.*

Algorithm 4 also satisfies similar properties of Lemmas 16-23 (Lemmas 26-33) and it can be also shown to be a self-stabilizing $\mathcal{L}$-algorithm by using these lemmas and the following Lemma 25. In Algorithm 3, two color pairs $\{A,C\}$ and $\{B,D\}$ of r and s cannot be reached from any initial configuration with same colors (Lemma 19). However, it cannot achieve Rendezvous from the initial configuration $\{A,C\}$ or $\{B,D\}$ (Lemma 14), since repetitions of $\{A,C\}$ and $\{B,D\}$ never attain Rendezvous. This is the reason why Algorithm 3 is not self-stabilizing. On the other hand, we can show that Algorithm 4 is self-stabilizing. In fact, it can solve Rendezvous from the initial configurations $\{A,C\}$, $\{B,D\}$, $\{C,E\}$, $\{D,A\}$ or $\{E,B\}$ as expressed in the following Lemma 25. Even if these configurations repeat, since the repetition contains $\{C,E\}$, Rendezvous succeeds. Otherwise, any configuration can reach some configuration $\{\alpha,\beta\}$ ($\alpha \to \beta$).

▶ **Lemma 25.** *Let $\{\ell(r,t_c),\ell(s,t_c)\} = \{\alpha,\gamma\}$, where $\alpha \to \beta$ and $\beta \to \gamma$. If Algorithm 4 starts with from any configuration $\{\ell(r,t_c),\ell(s,t_c)\} = \{\alpha,\gamma\}$, for any schedule of two robots after t_c, there is a cs-time $t^*(\geq t_c)$ such that*
$dis(p(r,t^*),p(s,t^*)) = 0$, or $dis(p(r,t^*),p(s,t^*)) \leq dis(p(r,t_c),p(s,t_c)) - \delta$
and $\{\ell(r,t^*),\ell(s,t^*)\} = \{\alpha',\beta'\}$ for some α' and β' ($\alpha' \to \beta'$).

The remaining lemmas can be proved similar to Lemmas 16-23.

▶ **Lemma 26.** *Let $\ell(r,t_c) = \ell(s,t_c)$ in Algorithm 4. If all LC-operations of r and s are performed at the same times, and let $t_i (i = 1,2,3,\dots)$ be the times r and s perform LC-operations simultaneously. Then $\ell(r,t_i) = \ell(s,t_i)$ for any time $t_i (i = 1,2,3,\dots)$ and there is a cs-time $t^*(\geq t_c)$ such that $dis(p(r,t^*),p(s,t^*)) = 0$.*

▶ **Lemma 27.** *Let $\ell(r,t_c) = \ell(s,t_c)$ in Algorithm 4. Assume that there are different times at which LC-operations of r and s are performed. Let t^* be the first cs-time after the first different time at which different LC-operations of r and s are performed. Then there are colors α^* and β^* such that $\{\ell(r,t^*),\ell(s,t^*)\} = \{\alpha^*,\beta^*\}$ and $\alpha^* \to \beta^*$.*

▶ **Lemma 28.** *Assume that Algorithm 4 starts with $\{\ell(r,t_c),\ell(s,t_c)\} = \{\alpha,\beta\}$ such that $\alpha \to \beta$. Let t^* be the first cs-time after t_c. Then there are colors α^* and β^* such that $\{\ell(r,t^*),\ell(s,t^*)\} = \{\alpha^*,\beta^*\}$ and $\alpha^* \to \beta^*$.*

▶ **Lemma 29.** *Let $\ell(r,t_c) = \ell(s,t_c)$. If Algorithm 4 is performed starting from t_c, there exist no cs-time $t^*(\geq t_c)$ such that $\{\ell(r,t^*),\ell(s,t^*)\} = \{\alpha,\gamma\}$, where $\alpha \to \beta$ and $\beta \to \gamma$.*

▶ **Lemma 30.** *If $dis(p(r,t_c),p(s,t_c)) = 0$ and Algorithm 4 is executed starting from t_c, $dis(p(r,t),p(s,t)) = 0$ for any $t \geq t_c$.*

▶ **Lemma 31.** *Let $\alpha = B$ and $\beta = C$, $\alpha = D$ and $\beta = E$, or $\alpha = E$ and $\beta = A$. If $\ell(r,t_c) = \alpha$ and $\ell(s,t_c) = \beta$ in Algorithm 4, then $\ell(s,t) = \beta$ and $p(s,t) = p(s,t_c)$ for any $t(t_c \leq t \leq t_c^+(r,LC))$.*

▶ **Lemma 32.** *If Algorithm 4 starts with $\{\ell(r,t_c),\ell(s,t_c)\} = \{B,C\}$, for any schedule of two robots after t_c, there is a cs-time $t^*(\geq t_c)$ such that $dis(p(r,t^*),p(s,t^*)) = 0$, or $dis(p(r,t^*),p(s,t^*)) \leq dis(p(r,t_c),p(s,t_c)) - \delta$ and $\{\ell(r,t^*),\ell(s,t^*)\} = \{C,D\}$*
or $\{\ell(r,t^*),\ell(s,t^*)\} = \{D,A\}$.

▶ **Lemma 33.** *If Algorithm 4 starts with $\{\ell(r,t_c),\ell(s,t_c)\} = \{\alpha,\beta\}$ such that $\alpha \to \beta$, for any schedule of two robots after t_c, there is a cs-time $t^*(\geq t_c)$ such that $\{\ell(r,t^*),\ell(s,t_c)\} = \{B,C\}$ and $dis(p(r,t^*),p(s,t^*)) \leq dis(p(r,t_c),p(s,t_c))$.*

Lemmas 25-33 follow the next theorem.

▶ **Theorem 34.** *Rendezvous is solved by SS-Rendezvous-with-5-colors(LC-atomic ASYNC, Non-Rigid, any). It is a self-stabilizing $\mathcal{L}$-algorithm.*

6 Concluding Remarks

We have shown that Rendezvous can be solved by $\mathcal{L}$-algorithms in LC-atomic ASYNC with the optimal number of colors of external-lights in the following cases. (1) Rigid and non-quasi-self-stabilizing, (2) Non-Rigid and quasi-self-stabilizing, and (3) Non-Rigid and self-stabilizing. We have also shown impossibility result that Rendezvous cannot be solved by any $\mathcal{L}$-algorithm with any number of colors of external-lights in LC-1-Bounded ASYNC. Combining it with our algorithms in LC-atomic ASYNC, we have shown that LC-atomic ASYNC is the maximal subclass in ASYNC Rendezvous can be solved by $\mathcal{L}$-algorithms with external-lights.

References

1. N. Agmon and D. Peleg. Fault-tolerant gathering algorithms for autonomous mobile robots. *SIAM J. Comput.*, 36(1):56–82, 2006.
2. Z. Bouzid, S. Das, and S. Tixeuil. Gathering of mobile robots tolerating multiple crash Faults. In *33rd ICDCS*, pages 334–346, 2013.
3. M. Cieliebak, P. Flocchini, G. Prencipe, and N. Santoro. Distributed computing by mobile robots: Gathering. *sicomp*, 41(4):829–879, 2012.
4. S. Das, P. Flocchini, G. Prencipe, N. Santoro, and M. Yamashita. Autonomous mobile robots with lights. *Theoretical Computer Science*, 609:171–184, 2016.
5. X. Défago, M. Gradinariu Potop-Butucaru, J. Clément, S. Messika, and P. Raipin Parvédy. Fault and Byzantine tolerant self-stabilizing mobile robots gathering - Feasibility study. CoRR abs/1602.05546, arXiv, 2016.
6. B. Degener, B. Kempkes, T. Langner, F. Meyer auf der Heide, P. Pietrzyk, and R. Wattenhofer. A tight run-time bound for synchronous gathering of autonomous robots with limited visibility. In *23rd ACM SPAA*, pages 139–148, 2011.
7. Y. Dieudonné and F. Petit. Self-stabilizing gathering with strong multiplicity detection. *tcs*, 428(13):47–57, 2012.
8. P. Flocchini, G. Prencipe, and N. Santoro. *Distributed Computing by Oblivious Mobile Robots*. Morgan & Claypool, 2012.
9. P. Flocchini, G. Prencipe, N. Santoroand, and P. Widmayer. Gathering of asynchronous robots with limited visibility. *tcs*, 337(1–3):147–169, 2005.
10. P. Flocchini, N. Santoro, G. Viglietta, and M. Yamashita. Rendezvous with Constant Memory. *tcs*, 621:57–72, 2016.
11. A. Hériban, X. Défago, and S. Tixeuil. Optimally gathering two robots. In *19th ICDCN:3*, pages 1–10, 2018.
12. T. Izumi, Z. Bouzid, S. Tixeuil, and K. Wada. Brief Announcement: The BG-simulation for Byzantine mobile robots. In *25th DISC*, pages 330–331, 2011.
13. T. Izumi, Y. Katayama, N. Inuzuka, and K. Wada. Gathering Autonomous Mobile Robots with Dynamic Compasses: An Optimal Result. In *21st DISC*, pages 298–312, 2007.
14. T. Izumi, S. Souissi, Y. Katayama, N. Inuzuka, X. Défago, K. Wada, and M. Yamashita. The gathering problem for two oblivious robots with unreliable compasses. *sicomp*, 41(1):26–46, 2012.
15. S. Kamei, A. Lamani, F. Ooshita, and S. Tixeuil. Asynchronous mobile robot gathering from symmetric configurations without global multiplicity detection. In *18th SIROCCO*, pages 150–161, 2011.
16. J. Lin, A.S. Morse, and B.D.O. Anderson. The multi-agent rendezvous problem. parts 1 and 2. *sicomp*, 46(6):2096–2147, 2007.
17. T. Okumura, K. Wada, and Y. Katayama. Brief Announcement: Optimal asynchronous Rendezvous for mobile robots with lights. In *19th SSS*, pages 484–488, 2017.
18. G. Prencipe. Impossibility of gathering by a set of autonomous mobile robots. *tcs*, 384(2–3):222–231, 2007.
19. S. Souissi, X. Défago, and M. Yamashita. Using eventually consistent compasses to gather memory-less mobile robots with limited visibility. *ACM Transactions on Autonomous and Adaptive Systems*, 4(1):1–27, 2009.
20. I. Suzuki and M. Yamashita. Distributed anonymous mobile robots: Formation of geometric patterns. *sicomp*, 28:1347–1363, 1999.
21. G. Viglietta. Rendezvous of two robots with visible bits. In *ALGOSENSORS 2013*, pages 291–306, 2014.

Linear Rendezvous with Asymmetric Clocks

Jurek Czyzowicz[1]
Département d'informatique, Université du Québec en Outaouais, Gatineau, Canada

Ryan Killick[2]
School of Computer Science, Carleton University, Ottawa, Canada

Evangelos Kranakis[3]
School of Computer Science, Carleton University, Ottawa, Canada

Abstract

Two anonymous robots placed at different positions on an infinite line need to rendezvous. Each robot possesses a clock which it uses to time its movement. However, the robot's individual parameters in the form of their walking speed and time unit may or may not be the same for both robots. We study the feasibility of rendezvous in different scenarios, in which some subsets of these parameters are not the same. As the robots are anonymous, they execute the same algorithm and when both parameters are identical the rendezvous is infeasible. We propose a universal algorithm, such that the robots are assured of meeting in finite time, in any case when at least one of the parameters is not equal for both robots.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms, Theory of computation → Distributed algorithms

Keywords and phrases anonymous, asymmetric clock, infinite line, rendezvous, mobile robot, speed, competitive ratio

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.25

1 Introduction

Rendezvous is concerned with two robots arbitrarily placed in a known search region and moving about until they meet each other. In this paper we will study symmetric rendezvous in which the two robots are instructed to employ the same algorithm. In our setting the robots' environment is an infinite line on which each robot may move at a constant speed. Each robot is equipped with its own clock which is used to time its movements and this clock is not necessarily consistent between the robots.

The rendezvous problem was studied for numerous models and various types of environments in randomized as well as deterministic settings. The fundamental question related to deterministic rendezvous concerns feasibility, or, more exactly, to identify the parameters of the model for which the rendezvous is possible to achieve (in finite time). The main concern related to the feasibility of rendezvous is that of *symmetry breaking*. Typical example of symmetry breaking is the use of the robot label in which the robot is aware of its label and may use its value as a parameter.

In the present paper the symmetry is broken yet another way. If the robots differ according to their speeds or private time units, we have a universal algorithm which guarantees rendezvous. Furthermore, and contrary to the case of labeled robots, knowledge as to which

[1] Research supported in part by NSERC Discovery grant
[2] Research supported by the Ontario Graduate Scholarship. Eligible for best student paper award.
[3] Research supported in part by NSERC Discovery grant

© Jurek Czyzowicz, Ryan Killick, and Evangelos Kranakis;
licensed under Creative Commons License CC-BY
22nd International Conference on Principles of Distributed Systems (OPODIS 2018).
Editors: Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira; Article No. 25; pp. 25:1–25:16

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

of the parameters is different is not necessary. Our robot is completely unaware of the value(s) of its individual parameters and it does not use them in the computations needed to run the algorithm.

When the rendezvous is feasible, research is concerned with the efficiency of the algorithm, which is usually measured by the time required until the meeting of the two robots takes place. The objective is to design algorithms that achieve good *competitive ratios* for the time spent by the robots to rendezvous divided by the time spent by the robots if they were running an optimal algorithm.

1.1 Model

We consider the symmetric rendezvous problem of two mobile robots $\mathcal{R}$ and $\mathcal{R}'$ modeled as points on the infinite line. The robots are initially located an unknown distance d from each other and the rendezvous problem is solved if it ever happens that the robots occupy the same position on the line at the same time (i.e. their trajectories intersect). The robots cannot see each other and must employ the same algorithm in order to rendezvous. We assume that robots can store and compute real numbers with arbitrary precision.

We consider a model in which each robot has its own constant speed and in which each is equipped with a clock allowing them to measure their travel time. Each robot will consider itself as the origin of its own coordinate system and it will use its clock to fix the distance unit for this coordinate system as the product of its maximum speed and local time unit. We explicitly consider the possibility that the robots have different speeds and / or clocks. We study algorithms which progress in a synchronous and continuous time model (i.e. robots are always active).

Without loss of generality, we will present our analysis from the viewpoint of the robot $\mathcal{R}$ and thus assume that this robot has maximum unit speed, and that its clock is "correct" in the sense that it agrees with some predefined global coordinate system. On the other hand, we set the speed of $\mathcal{R}'$ as $v > 0$, and set its time unit as $\tau > 0$ with the result that one time unit as measured by the clock of $\mathcal{R}'$ will actually be τ time units as measured by the clock of $\mathcal{R}$. The robots will determine their progress/distance traveled in an algorithm as the product of their travel time and maximum speed.

We specifically focus on three sub-models obtained from the preceding general model by fixing one of v, τ, or the product $v\tau$ to one. In the equal time-unit model (or T-Model) $\tau = 1$, in the equal distance-unit model (or D-Model) $v\tau = 1$, and in the equal speeds model (or V-Model) $v = 1$. Since only one of v or τ is independent in these models we will assume without loss of generality that $0 < v < 1$ in the T- and D-Models and take $0 < \tau < 1$ in the V-Model.

In analyzing the time complexity of our rendezvous algorithms we will employ an adversarial argument in which we assume that an adversary is able to choose the values of d, v, and/or τ in order to maximize the *competitive ratio* of a given algorithm (i.e. we employ a worst-case analysis). Loosely defined, the competitive ratio of a rendezvous algorithm $\mathcal{A}$ is the maximum ratio of the time it takes the robots to rendezvous using $\mathcal{A}$ divided by the time it would take them given that they are running an optimal algorithm. We will give a more precise definition of the competitive ratio at a later time.

To end this section we observe that to be most general one should consider the possibility that the robots also differ in their orientations (i.e. sense of positive direction). However, in all cases for rendezvous on the line studied here, having different orientations will only help the robots in achieving their goal. As a result, all derived upper bounds will not be affected by a difference in orientation and, since including this only serves to complicate notation, we will not explicitly consider it in this work.

1.2 Related Work

Search may be viewed as a game between two players having divergent goals, one trying to hide for as long as possible and the other one attempting to minimize its search time. As a contrast, in rendezvous the two involved players have converging goals in that they are aiming to find one another as quickly as possible. The duality of the two problems has been presented and investigated in the beautiful book [1].

The search problem for one robot on an infinite line was initiated independently by Bellman [7] and Beck [5]. There have been numerous studies on search even for an environment as simple as the infinite line, emphasizing various aspects arising from the capabilities of one or more robots and the status of the search domain. These include randomized [23], group search [10], linear terrains [14], faulty searchers [15], and turn costs [17]. The efficiency of linear search is most often measured by the *competitive ratio*, which is the time spent by the robot to complete its search divided by the time needed by an omniscient robot that knows the location of the target. The competitive ratio of 9 is obtained by the *cow-path* algorithm (cf. [7] and Beck [5]) and was first proved to be optimal for stochastic linear search in [6] and deterministically in [2, 3].

The approach to solving the rendezvous problem is most often fundamentally different from the techniques applied for search (although, somewhat surprisingly, this will not always be the case in the present paper). The solution of the symmetric rendezvous problem requires that the two robots are somehow equipped to break symmetry [25]. There has been extensive research literature concerned with taking advantage of "innate" asymmetries of the studied model. For example, [16, 26, 18, 19] focus on robots having distinct labels, [24, 27] on robots equipped with identical tokens that can be placed on selected nodes, [12] on a robot's awareness of its GPS position in the environment.

The rendezvous (and its more general version of gathering) problem has been also studied for robots of different speeds [8, 20], inconsistent compasses [11, 22] and chirality or sense of direction [4, 9]. However, in the studies previously mentioned, these differences were obstacles that needed to be circumvented by the suggested algorithms, rather than used for the benefit of the proposed approach, which is the case of the present paper. To the best of our knowledge, linear rendezvous for robots with asymmetric clocks has never been studied before.

1.3 Results and outline

In this paper we study the rendezvous problem when the robots are equipped with asymmetric clocks. In addition to exploring novel ways of defining rendezvous algorithms, we demonstrate the feasibility of rendezvous and give two algorithms solving rendezvous in finite time provided that at least one of a robot's maximum speed or time unit differs from the other (see end of Section 5). In addition, we study rendezvous in the three restricted models, T-model, D-model, and V-model.

In Section 3 we analyze rendezvous in the T-Model and show that possessing equal time-units reduces rendezvous into the problem of search, allowing one to optimally solve the problem with a competitive ratio of 9. In Section 4 we analyze rendezvous under the assumption that the robots have equal distance-units, i.e. $v\tau = 1$. Here we get our first taste of the difficulties involved with asymmetric clocks. We show that rendezvous is solved with a competitive ratio of $\frac{105}{11} \approx 9.55$. Furthermore, in the limit of large d, the competitive ratio is 9.

In Section 5 we analyze rendezvous when the robots' speeds are equal and $0 < \tau < 1$. This is the most difficult model to analyze and we will observe large differences in the algorithms employed and resulting upper bounds, as compared to the T- and D-Models. We give two algorithms that solve rendezvous, the first with a competitive ratio of $\mathcal{O}\left(\frac{\tau \log^2(d)}{\log(\tau)}\right)$ and the second with a slightly tunable competitive ratio of $\mathcal{O}\left(\frac{\tau \log^{1+c}(d)}{c \log(\tau)}\right)$, where $c > 0$ is a parameter of the algorithm. We also offer arguments as to why it may not be possible to achieve a constant competitive ratio in this model.

2 Preliminaries and Notation

In this section we introduce some preliminary ideas and notations used throughout our analysis. We begin with some notation.

As usual, for any real number we use $|\cdot|$ to indicate its absolute value, $\log(\cdot)$ to indicate the base-2 logarithm, and $\ln(\cdot)$ to indicate the natural logarithm with base e.

2.1 Time and position space

It will be useful to consider rendezvous algorithms as specifying a piecewise-continuous trajectory in a two dimensional space where the horizontal axis represents a robot's position on the line and the vertical axis represents the flow of time. In this representation one can view the effect of v, τ, and d as a scaling and translation of the coordinate axes of the local xt-space of $\mathcal{R}'$ as compared to $\mathcal{R}$. As a result, if we represent the trajectory $t(x)$ of $\mathcal{R}$ as a (possibly multivalued) function of position on the line, then the actual trajectory of $\mathcal{R}'$ will be $\tau \cdot t\left(\frac{x \pm d}{v\tau}\right)$. Equivalently, if the trajectory of $\mathcal{R}$ is represented as a function of time $x(t)$, then the trajectory of $\mathcal{R}'$ is $v\tau x\left(\frac{t}{\tau}\right) \pm d$. It would help to remember these transformations as they will be used repeatedly throughout our analysis.

2.2 Rendezvous algorithms

In what follows assume that we are speaking about the robot $\mathcal{R}$.

We consider algorithms whereby robots move between an infinite sequence of *turning-points* $P_k = (X_k,\ T_k)$ which specify the time and location on the line at which a robot reverses its direction. On the round k of such an algorithm a robot will move from its initial position to the turning-point P_k and then return to its initial position. The time necessary to finish a round will be $2|X_k|$ and the time at which a robot begins the round k is $2\sum_{i=0}^{k-1}|X_i|$. The time T_k at which a robot is at the turning-point P_k can thus be written as

$$T_k = 2\sum_{i=0}^{k-1}|X_i| + |X_k|. \tag{1}$$

Since T_k is dependent on X_k we will often refer to X_k as the kth turning-point. We will also adopt the convention that P_{-1} indicates a robot's starting location and assume without loss of generality that $X_k > 0$ when k is even (i.e. a robot initially moves to the right). Finally, we will always explicitly state whether or not $\mathcal{R}'$ begins on the left or right of $\mathcal{R}$ and assume that $d > 0$.

Using the same terminology as [1] we call an infinite sequence of turning-points periodic and monotonic if, for all $k \geq 0$, it satisfies

$$X_{2k-1} < X_{2k+1} < \cdots < 0 < \cdots < X_{2k} < X_{2k+2}. \tag{2}$$

Algorithm 1 Rendezvous[X_k].

1: $k = 0$
2: **repeat**
3: Move to the position X_k and return to initial position. $k = k + 1$.
4: **until** Meeting occurs

As discussed[4] in [1], an adversary may achieve an arbitrarily large competitive ratio if an algorithm ever has a first turning-point. This results from the ability of an adversary to choose d arbitrarily small with respect to this first turning-point and force $\mathcal{R}$ to initially move away from the initial position of $\mathcal{R}'$. As a result, one either needs to consider doubly infinite sequences of turning-points or, more practically, make the additional assumption that the robots possess knowledge of a lower bound on d (which might be indirectly due to the robot having a finite size or visibility range). In this work we are primarily interested in large values of d and / or v and τ close to one, we will take the latter approach and assume that the adversary is restricted to values of d that are comparable in size to the first turning-point of a rendezvous algorithm.

We observe that any periodic and monotonic algorithm will have turning-points that span a cone-shaped curve $\mathcal{T}(x)$ in xt-space satisfying the identity $T_k = \mathcal{T}(X_k)$ (i.e. the turning points lie on the boundary of a cone-shaped curve). This observation allows us to give an alternate, and particularly useful representation of a rendezvous algorithm – we specify the curve $\mathcal{T}(x)$ and have the robots infer where their turning-points are located. Using this method a robot will be instructed to move in one direction until its trajectory in xt-space intersects the curve $\mathcal{T}(x)$. It will then reverse its direction and move at full speed until its trajectory again intersects $\mathcal{T}(x)$, and so on. The robots can compute their turning-points for this type of algorithm using the relation $T_k = \mathcal{T}(X_k)$.

If an algorithm is given by its turning-points X_k then we say that X_k induces the curve $\mathcal{T}(x)$. If an algorithm is specified by $\mathcal{T}(x)$ then we say that $\mathcal{T}(x)$ induces the turning-points X_k. We call an algorithm symmetric if the curve $\mathcal{T}(x)$ is an even function of x. We will be interested in algorithms that are symmetric, periodic, and monotonic and we call SPM the class of all such algorithms. We will use the following formal definition, based on $\mathcal{T}(x)$, for an algorithm to be in the class SPM:

▶ **Definition 1.** An algorithm with turning-points X_k is in the class SPM if the curve $\mathcal{T}(x)$ which induces these turning-points satisfies:

1. $\mathcal{T}(x)$ is an even function, i.e. $\mathcal{T}(x) = \mathcal{T}(-x)$.
2. There exists an $X_0 > 0$ such that $X_0 = \mathcal{T}(X_0)$.
3. For all $|x| > X_0$, $\mathcal{T}(x) > |x|$.
4. $\mathcal{T}(x)$ is continuously differentiable for all $x > X_0$.

One can easily confirm that if $\mathcal{T}(x)$ satisfies the above definition, the induced turning-points will be periodic and monotonic (as per the condition (2)). In the sequel we will construct algorithms which belong to the class SPM.

We formally define Algorithm 1 and Algorithm 2 which respectively take X_k and $\mathcal{T}(x)$ as parameters. So far we have only discussed rendezvous algorithms from the perspective of $\mathcal{R}$. When referring to the turning-points etc. of $\mathcal{R}'$ we will indicate this using a prime. So,

[4] Technically this was discussed for the case of search but the argument also applies here.

Algorithm 2 Rendezvous[$\mathcal{T}(x)$].

1: Run Algorithm 1 with X_k defined by $T_k = \mathcal{T}(X_k)$.

for example, the turning-points of $\mathcal{R}'$ will be $P'_k = (X'_k,\ T'_k)$ and the curve induced by these turning-points will be $\mathcal{T}'(x)$. We note that $X'_k = v\tau X_k \pm d$, $T'_k = \tau T_k$, and $\mathcal{T}'(x) = \tau\mathcal{T}\left(\frac{x\mp d}{v\tau}\right)$.

2.3 Robot trajectories and rendezvous points

Viewed as a whole, a rendezvous algorithm will specify a trajectory in xt-space composed of a series of line segments connected at their endpoints. For an algorithm defined by the sequence X_k we can explicitly write the equations of the lines traversed by $\mathcal{R}$ as

$$t_k(x) = (-1)^k x + T_k - |X_k|. \tag{3}$$

where $t_k(x)$ is the segment beginning at P_{k-1} and ending at P_k. Likewise, the lines traversed by $\mathcal{R}'$ can be written as

$$t'_k(x) = (-1)^k \frac{x \mp d}{v} + \tau T_k - \tau|X_k|. \tag{4}$$

We claim the following:

▶ **Lemma 2.** *Assume that we have chosen $|X_k|$ such that Algorithm 1 solves rendezvous. Then, if the robots meet when $\mathcal{R}$ is approaching its kth turning-point and $\mathcal{R}'$ is approaching its jth turning-point then the time of rendezvous can be written as*

$$t_{k,j} = \frac{\pm(-1)^k d - (-1)^{k+j} v\tau(T_j - |X_j|) + T_k - |X_k|}{1 - (-1)^{k+j} v}$$

Proof. It is obvious that the possible points of rendezvous will occur at intersections of pairs of lines $t_k(x)$ and $t'_j(x)$, $k, j \geq 0$. The lemma follows from solving the equation $t_k(x) = t'_j(x)$. ◀

2.4 Competitive ratios

We are interested in algorithms that achieve small competitive ratios where we have defined the competitive ratio of an algorithm $\mathcal{A}$ as the supremum ratio of the time it takes to rendezvous using $\mathcal{A}$ to the time taken to rendezvous if the robots employ an optimal algorithm. We now give more precise definitions.

▶ **Definition 3.** Let $\mathcal{A}$ be an algorithm solving rendezvous in the T- or D-Models and let the time of rendezvous be $T_*(v,d)$. Then the competitive ratio of $\mathcal{A}$ is $CR = \sup_{v,d} \frac{(1-v)T_*(v,d)}{d}$.

▶ **Definition 4.** Let $\mathcal{A}$ be an algorithm solving rendezvous in the V-Model and let the time of rendezvous be $T_*(\tau,d)$. Then the competitive ratio of $\mathcal{A}$ is $CR = \sup_{\tau,d} \frac{(1-\tau)T_*(\tau,d)}{d}$.

We will justify these definitions as we consider each model in turn.

2.5 Feasibility of rendezvous

Here we establish the feasibility of rendezvous. To this end consider an algorithm in the class SPM with the curve $\mathcal{T}(x)$. We say that $\mathcal{T}(x)$ *contains* $\mathcal{T}'(x)$ (resp. $\mathcal{T}'(x)$ contains $\mathcal{T}(x)$) if there exists an x_0 such that for all x satisfying $|x| > |x_0|$ we have $\mathcal{T}(x) \leq \mathcal{T}'(x)$ (resp.

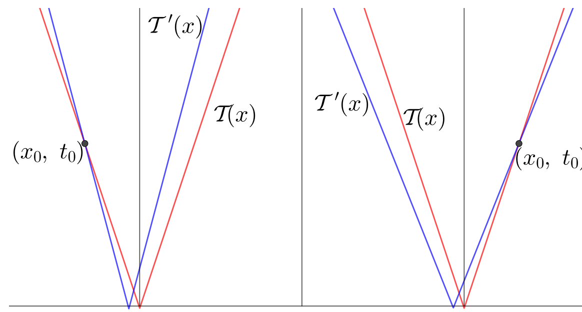


Figure 1 Illustration of the idea of containment. On the left $\mathcal{T}(x)$ contains $\mathcal{T}'(x)$ and on the right $\mathcal{T}'(x)$ contains $\mathcal{T}(x)$. In both cases the containment point is indicated.

$\mathcal{T}(x) \geq \mathcal{T}'(x)$). If such an x_0 exists we call x_0 and $t_0 = \mathcal{T}(x_0)$ the *containment point* and *containment time* of $\mathcal{T}(x)$ and $\mathcal{T}'(x)$. Intuitively, x_0 will be an intersection point of $\mathcal{T}(x)$ and $\mathcal{T}'(x)$. Figure 1 illustrates these definitions. We can now claim the following:

▶ **Lemma 5.** *If either of $\mathcal{T}(x)$ or $\mathcal{T}'(x)$ contains the other, then rendezvous is guaranteed.*

Proof. Assume that $\mathcal{T}(x)$ contains $\mathcal{T}'(x)$ and let t_0 be the containment time. Let X_k be the first turning-point that $\mathcal{R}$ reaches after the time t_0 and assume that $X_k > 0$. In this situation $\mathcal{R}$ will be on the right of $\mathcal{R}'$ once it reaches X_k and will be on the left of $\mathcal{R}'$ once it reaches X_{k+1}. Likewise, if $X_k < 0$, then $\mathcal{R}$ will be on the left of $\mathcal{R}'$ at X_k and on the right of $\mathcal{R}'$ at X_{k+1}. In either case the robots must rendezvous between the turning-points X_k and X_{k+1} of $\mathcal{R}$. In a similar manner one can confirm that the robots will rendezvous between the turning-points X'_j and X'_{j+1} of $\mathcal{R}'$ where X_j is the first turning-point $\mathcal{R}'$ reaches after the time t_0. ◀

In both the T- and D-models there are a wide range of algorithms in the class SPM with curves $\mathcal{T}(x)$ such that $\mathcal{T}(x)$ contains $\mathcal{T}'(x)$. In particular, if $v \neq 1$ and $\mathcal{T}(x)$ is a linear function, then $\mathcal{T}'(x) = \tau\mathcal{T}\left(\frac{x \mp d}{v\tau}\right) = \frac{1}{v}\mathcal{T}(x \mp d)$ and $\mathcal{T}(x)$ will clearly contain this. If $v = 1$ then there are still a variety of curves one can choose (for example, $\mathcal{T}(x) = x^2$) and thus rendezvous is guaranteed in general.

3 T-Model

In this section we analyze rendezvous under the assumption that the robots have the same time units, i.e. $\tau = 1$. This assumption turns out to be rather powerful as it allows us to reduce the problem into the problem of search for a stationary target.

▶ **Theorem 6.** *Rendezvous in T-Model is equivalent to search for a stationary target at distance $d_* = \frac{d}{1-v}$.*

Proof. Assume that we have a rendezvous algorithm that specifies the trajectory $R(t)$ for $\mathcal{R}$. The robot $\mathcal{R}'$ will then follow the actual trajectory $R'(t) = vR(t) \pm d$. Now consider a coordinate system moving with the robot $\mathcal{R}'$ and scaled by a factor of $\frac{1}{1-v}$. In this coordinate system $\mathcal{R}'$ will appear to be stationary at the position $\pm\frac{d}{1-v}$ and $\mathcal{R}$ will appear to move along the trajectory $R_*(t) = R(t)$. We can thus view this problem as a search problem for a target at distance $\frac{d}{1-v}$ and any algorithm solving search will also solve rendezvous in the same amount of time. ◀

This clearly justifies our definition of the competitive ratio for this model. Furthermore, this result allows us to draw on many of the results known about search and, in particular, allows one to optimally solve the rendezvous problem.

▶ **Theorem 7.** *Rendezvous in T-Model is optimally solved with a competitive ratio of* 9 *using Algorithm 1 with* $|X_k| = 2^k$.

Proof. Observe that Algorithm 1 with $|X_k| = 2^k$ is the familiar cow-path algorithm which optimally solves search for a target at distance d in time $9d$. Since search is equivalent to rendezvous in T-Model, this algorithm will optimally solve rendezvous as well. ◀

Note that the optimal solution in this case lies within the class SPM.

4 D-Model

In this section we analyze rendezvous under the assumption that the robots have the same distant-units, i.e. $v\tau = 1$. Unlike the T-model, we cannot transform the problem into the search problem and thus we will approach our analysis slightly different. We begin with a lemma that justifies our definition of the competitive ratio for this model.

▶ **Theorem 8.** *Rendezvous in D-model takes time at least* $\frac{d}{1-v}$.

Proof. Consider any rendezvous algorithm with turning points X_k. Assume that the robots rendezvous between the $(k-1)^{st}$ and k^{th} turning points of $\mathcal{R}$ and the $(j-1)^{st}$ and j^{th} turning points of $\mathcal{R}'$. Furthermore, assume that $\mathcal{R}'$ begins to the right of $\mathcal{R}$. In this case $\mathcal{R}$ must be moving to the right when the robots rendezvous and thus k will be even. Assume that d and τ are chosen such that the robots rendezvous when j is also even. Then, by Lemma 2, the robots will rendezvous at the time

$$t_{k,j} = \frac{d - (T_j - |X_j|) + (T_k - |X_k|)}{1-v} = \frac{d - 2\sum_{i=0}^{j-1}|X_i| + 2\sum_{i=0}^{k-1}|X_i|}{1-v}.$$

Since $v < 1$ and $v\tau = 1$ we have $\tau > 1$ and thus $\mathcal{R}'$ will always take longer to finish a round. We must therefore have $j \leq k$ and the rendezvous time will be at least $\frac{d}{1-v}$. ◀

We note that the above lower bound is general and applies to any algorithm (not just those in the class SPM). We now claim the following:

▶ **Theorem 9.** *Rendezvous in D-Model is solved with a competitive ratio of* $\frac{105}{11}$ *using Algorithm 1 with* $|X_k| = 2^k$. *Furthermore, in the limit of large d, the competitive ratio is* 9.

Proof. Assume that $\mathcal{R}$ is approaching its kth turning-point and when $\mathcal{R}'$ is approaching its jth turning-point the robots rendezvous. Then, by Lemma 2, they will rendezvous at the time

$$t_{k,j} = \frac{\pm(-1)^k d - (-1)^{k+j}(T_j - |X_j|) + T_k - X_k}{1-(-1)^{k+j}v}.$$

Observe that the robots will never rendezvous when $\mathcal{R}$ is traveling away from $\mathcal{R}'$. Thus, if $\mathcal{R}'$ begins to the right of $\mathcal{R}$, k must be even. Likewise, if $\mathcal{R}'$ begins to the left of $\mathcal{R}$, k must be odd. On the other hand, $\mathcal{R}'$ may be moving towards or away from $\mathcal{R}$ when they rendezvous and thus j may be even or odd. This gives two possible rendezvous times depending on the parity of $k+j$. If $k+j$ is even then $t_{k,j} = \frac{d+2^{k+1}-2^{j+1}}{1-v}$ and if $k+j$ is odd then

$t_{k,j} = \frac{d+2^{k+1}+2^{j+1}-4}{1+v}$. The competitive ratios for these two cases are $CR_+ = 1 + \frac{2}{d}(2^k - 2^j)$ and $CR_- = \frac{1-v}{1+v}\left[1 + \frac{2}{d}(2^k + 2^j - 2)\right]$ where CR_+ (resp. CR_-) corresponds to $k+j$ even (resp. $k+j$ odd).

We note that, in order for $\mathcal{R}$ and $\mathcal{R}'$ to meet while traveling towards their k^{th} and j^{th} turning-points, the time $t_{k,j}$ must satisfy $T_{k-1} \leq t_{k,j} \leq T_k$ and $T'_{j-1} \leq t_{k,j} \leq T'_j$. Furthermore, since $\mathcal{R}'$ will take longer to finish a round, there can be at most one turning-point of $\mathcal{R}'$ between any two turning-points of $\mathcal{R}$.

Now assume that $k+j$ is even. In this case an adversary will get the best payoff if they choose d as small as they can without causing the robots to rendezvous on an earlier round. The smallest value of d is achieved if d and v can be chosen such that the $(k-2)^{nd}$ turning-point of $\mathcal{R}$ is arbitrarily close to the $(j-2)^{nd}$ turning-point of $\mathcal{R}'$ (see the left side of Figure 2). We claim that the adversary can choose d and v to achieve this. To see why assume that the adversary has chosen d and v such that $T_{k-2} = T'_{j-2}$. In order to rendezvous at the time $t_{k,j}$ then we need to satisfy $T'_{j-1} \leq t_{k,j} \leq T'_j$. Since the robots will not rendezvous when $\mathcal{R}$ is moving away from $\mathcal{R}'$, we can easily conclude that $T'_{j-1} \leq t_{k,j}$. Also, since $t_{k,j}$ must be smaller than T_k, we can also conclude that $t_{k,j} \leq T'_j$ due to the fact that $T'_{j-2} = T_{k-2}$ and because $\mathcal{R}'$ takes longer to finish a round. Thus, in the worst case, we may take $X_{k-2} = X'_{j-2}$ which, after simplification, tells us that we should take $d = 2^{k-2} - 2^{j-2}$. Using this result in the expression for CR_+ yields a competitive ratio of 9.

Now assume that $k+j$ is odd. In this case the competitive ratio will depend on both d and v. However, we can similarly conclude that the adversary would like to minimize d and thus they will try to choose d and v in order to make the $(j-1)^{st}$ turning-point of $\mathcal{R}'$ equal (or arbitrarily close) to the $(k-2)^{nd}$ turning-point of $\mathcal{R}$. However, in this case, we claim that the adversary cannot always do this. Indeed, if $T_{k-2} = T'_{j-1}$ and $X_{k-2} = X'_{j-1}$ then, after some manipulation, we find that,

$$2^{j-1} = \frac{dv}{1-v} + \frac{2}{3} \tag{5}$$

and

$$2^{k-2} = \frac{d}{1-v} + \frac{2}{3}. \tag{6}$$

Since we are assuming that $\mathcal{R}'$ meets $\mathcal{R}$ as it is traveling towards its jth turning-point we need $t_{k,j} \leq T'_j$. Using (6), (5), and the expression for $t_{k,j}$ we find that d must satisfy $d \leq \frac{2}{3}\left(\frac{1}{v} - 1\right)$ and thus j must satisfy $2^{j-1} = \frac{dv}{1-v} + \frac{2}{3} \leq \frac{4}{3}$. Clearly, the only j that satisfies this is $j = 1$ and we can conclude that, in the case that $k+j$ is odd, the worst-case situations occur when $j = 1$ (and k is even). When $j = 1$, we have from (5) that $d = \frac{1-v}{3v}$ and from (6) that $2^k = \frac{4}{3}\left(\frac{1}{v} + 2\right)$. Using these results in the expression for CR_- gives us $CR_- = \frac{15v+9}{1+v}$ and this can be seen to increase with v. However, we cannot simply maximize this over v. If we take $k = 2$ then we need $v = 1$ and $d \leq 0$ and this is clearly not possible. We can thus conclude that $k \geq 4$. If $k = 4$ then one can confirm that $v = \frac{1}{10}$, and if $k > 4$ then $v < \frac{1}{10}$. Thus, we find the maximum competitive ratio when $v = \frac{1}{10}$. Substituting this result in for CR_- we find that $CR_- \leq \frac{105}{11}$.

Finally, since we found that $2^k = \frac{4}{3}\left(\frac{1}{v} + 2\right)$ in the worst case, we can see that for very large k we will have v very small. Since k will be large for large d and since $CR_- = \frac{15v+9}{1+v} = 9$ when $v = 0$, we can conclude that the competitive ratio approaches 9 as d gets large. ◀

We note that the upper-bound of Theorem 9 is tight for the algorithm considered since it is easily confirmed that the competitive ratio is exactly 9.55 when $v = \frac{1}{10}$ and $d = 3$.

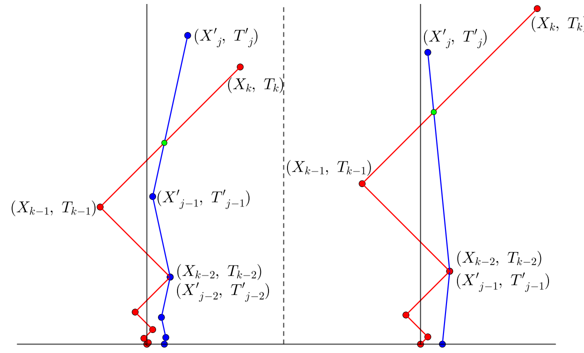


Figure 2 The two worst case possibilities in D-Model using Algorithm 1 with $|X_k| = 2^k$. The point of rendezvous is indicated in green and in both cases the robots rendezvous as $\mathcal{R}$ is approaching its jk^{th} turning-point and when $\mathcal{R}'$ is approaching its jth turning-point. Left: The case that $k + j$ is even. Right: The case that $k + j$ is odd. The absolute worst-case is depicted and occurs when $d = 3$ and $v = \frac{1}{10}$ such that the competitive ratio is 9.55. Note that the scales of the depicted trajectories are not the same for the two cases.

What is somewhat surprising about this result is that the competitive ratio is nearly identical to that of the T-Model when d is large, and this is achieved using the same algorithm. We suspect that this algorithm is optimal here since, in the limit that v goes to zero, this model reduces to search for which a competitive ratio of 9 is optimal. We do not have a formal proof of this, however, and thus it is still an open question whether or not this algorithm is optimal.

5 V-Model

In this section we analyze upper bounds on rendezvous when the robots' speeds are equal and $0 < \tau < 1$. In both the T-Model and D-Model the robots employed Algorithm 1 with a geometric sequence of turning-points ($|X_k| = 2^k$) in order to rendezvous with a small constant competitive ratio. We will see that the V-Model is rather more complicated, and, in particular, a geometric sequence of turning-points will not work (see Lemma 18 at the end of this section). We will therefore have to employ a different type of algorithm. We begin, however, with a lower-bound to justify our definition of the competitive ratio:

▶ **Theorem 10.** *Rendezvous in V-Model takes time at least $\frac{d}{1-\tau}$.*

Proof. For concreteness assume that $\mathcal{R}'$ begins to the right of $\mathcal{R}$. Then, the robots will rendezvous as $\mathcal{R}$ is traveling to the right and $\mathcal{R}'$ is traveling to the left. Consider the jth turning-point of $\mathcal{R}'$ and assume that j is even such that $\mathcal{R}'$ moves left after this turning-point. Let X_k be the first even turning-point of $\mathcal{R}'$ after the jth turning-point of $\mathcal{R}'$. Observe that the robots will rendezvous before the $(j+1)^{st}$ turning-point of $\mathcal{R}'$ provided that $X'_j - X_k \le T_k - T'_j$. Now assume that τ is close enough to one such that we may assume that $j = k$. In this case we can rewrite the condition $X'_j - X_k \le T_k - T'_j$ as $T_k + X_k \ge \frac{d}{1-\tau}$.

Since we are free to choose d and τ let us choose these parameters such that $T_k + X_k = \frac{d}{1-\tau} - \epsilon$ for an arbitrarily small $\epsilon > 0$. Then the robots will not rendezvous until some time after the $(k+2)^{nd}$ turning-point of $\mathcal{R}'$. Since $T_{k+2} = T_{k+1} + |X_{k+1}| + |X_{k+2}| = T_k + |X_k| + 2|X_{k+1}| + |X_{k+2}| = \frac{d}{1-\tau} - \epsilon + 2|X_{k+1}| + |X_{k+2}|$ we can conclude that the robots will take at least $\frac{d}{1-\tau}$ time to rendezvous. ◀

Note that this lower bound applies to any algorithm. Now for an upper bound. We claim the following:

▶ **Theorem 11.** *Rendezvous in V-Model is solved with a competitive ratio of* $\frac{18\log^2(d)\tau}{|\log(\tau)|} + \mathcal{O}\left(\frac{\log(d)\tau}{|\log(\tau)|}\right)$ *using Algorithm 1 with* $|X_k| = (k+2)2^k$.

An overview of the proof of this theorem is as follows. We first show that Algorithm 1 with $|X_k| = (k+2)2^k$ does indeed solve rendezvous and we will do this by demonstrating that $\mathcal{T}(x)$ contains $\mathcal{T}'(x)$. Since $\mathcal{T}(x)$ contains $\mathcal{T}'(x)$, the robots will be guaranteed to rendezvous by the second turning-point reached by $\mathcal{R}$ after the containment time and we will use this fact to bound the rendezvous time. Throughout the proof we will need to make use of the Lambert-W function (or simply Lambert function) $W(x)$ which is defined as the inverse function of $f(x) = xe^x$ (we consider the real valued branches only and thus x is restricted to the range $x \geq \frac{-1}{e}$). Since $W(x)$ is multivalued on the open interval $(-e^{-1}, 0)$ it is usual to define $W_{-1}(x)$ as the branch which attains values ≤ -1 and reserve the use of $W(x)$ to refer to the principal branch which attains values ≥ -1. We will need the following properties of $W(x)$ which are found in, or trivially derived from, the results in [21] and [13]:

▶ **Lemma 12.** *The two real valued branches* $W(x)$ *and* $W_{-1}(x)$ *satisfy:*

$$W(x) \leq \ln(x) - \ln(\ln(x)) + \frac{e}{e-1} \cdot \frac{\ln(\ln(x))}{\ln(x)}, \quad x \geq e \tag{7}$$

$$W(x) \geq \ln(x) - \ln(\ln(x)) + \frac{\ln(\ln(x))}{2\ln(x)}, \quad x \geq e \tag{8}$$

$$W_{-1}(x) < \ln(-x), \tag{9}$$

$$\frac{d}{dx}\frac{x}{W(x)} = \frac{1}{1+W(x)}, \tag{10}$$

$$\frac{d^2}{dx^2}\frac{x}{W(x)} < 0. \tag{11}$$

Before we can demonstrate that the algorithm solves rendezvous we need to first determine the curve $\mathcal{T}(x)$ induced by the turning-points $|X_k| = (k+2)2^k$.

▶ **Lemma 13.** *Let* $\mathcal{T}(x)$ *be the curve induced by Algorithm 1 with* $|X_k| = (k+2)2^k$. *Then*

$$\mathcal{T}(x) = 3|x| - \frac{4\ln(2)|x|}{W(4\ln(2)|x|)}.$$

Proof. One can observe that the turning-points $|X_k| = (k+2)2^k$ form an arithmetico-geometric sequence and its sum has the closed form expression $\sum_{i=0}^{k-1}|X_i| = k2^k$. We therefore have $T_k = 2\sum_{i=0}^{k-1}|X_i| + |X_k| = (3k+2)2^k$. We may rewrite T_k as $T_k = 3(k+2)2^k - 4\cdot 2^k = 3|X_k| - \frac{4|X_k|}{k+2}$.

To express T_k fully in terms of $|X_k|$ we need to invert $|X_k| = (k+2)2^k$. We can do this using the Lambert function. First rewrite the equation $|X_k| = (k+2)2^k$ as $4\ln(2)|X_k| = \ln(2)(k+2)e^{\ln(2)(k+2)}$. In this form we can directly apply the definition of the Lambert function to get the solution $\ln(2)(k+2) = W(4\ln(2)|X_k|)$. We can therefore express T_k as $T_k = 3|X_k| - \frac{4\ln(2)|X_k|}{W(4\ln(2)|X_k|)}$. Since $\mathcal{T}(X_k) = T_k$ the lemma follows. ◀

Now that we have determined the curve $\mathcal{T}(x)$ induced by the turning points $|X_k| = (k+2)2^k$, we can show that $\mathcal{T}(x)$ will contain $\mathcal{T}'(x)$ and thus the algorithm will solve rendezvous.

▶ **Lemma 14.** *Consider Algorithm 1 with* $|X_k| = (k+2)2^k$. *Then* $\mathcal{T}(x)$ *contains* $\mathcal{T}'(x)$.

Proof. We need to show that, for all d and $\tau < 1$, there exists an x_0 such that for all $|x| > x_0$ we have $\mathcal{T}'(x) > \mathcal{T}(x)$. To do this we will assume that $d = 0$ and show that the difference $D(x) = \mathcal{T}'(x) - \mathcal{T}(x)$ grows without bound for all τ satisfying $0 < \tau < 1$. If this is the case, then, no matter the value of d, there will eventually be an x_0 such that for all $x > x_0$ we have $\mathcal{T}'(x) > \mathcal{T}(x)$.

To this end consider the rate of change of $\mathcal{T}(x)$ for $x > 0$. Using (10) we find that $\frac{d\mathcal{T}(x)}{dx} = 3 - \frac{4\ln(2)}{1+W(4\ln(2)x)}$. Since $W(0) = 0$ and $W(x)$ is an increasing function, $\mathcal{T}(x)$ is also increasing for all $x > 0$. Furthermore, if $d = 0$ we have $\frac{d\mathcal{T}'(x)}{dx} = 3 - \frac{4\ln(2)}{1+W(4\ln(2)\frac{x}{\tau})}$ and this is clearly larger than $\frac{d\mathcal{T}(x)}{dx}$ for $0 < \tau < 1$. The rate of change of $D(x)$ must therefore always be positive and thus $D(x)$ does indeed grow without bound. ◀

We need one more simple lemma before tackling the proof of Theorem 11.

▶ **Lemma 15.** *Consider Algorithm 1 with $|X_k| = (k+2)2^k$ and let $\mathcal{T}(x)$, x_0, and t_0 be the induced curve, and containment time and position. Then $t_0 < 3x_0$ and $\tau\mathcal{T}(x-d) > \mathcal{T}(x)-3d$.*

Proof. The first part of the lemma follows easily from the fact that $\frac{d\mathcal{T}(x)}{dx} < 3$. The second part also follows easily from the facts that $\frac{d\mathcal{T}(x)}{dx} < 3$ and $\mathcal{T}(x)$ is a convex function (see (11)). As a result, $\mathcal{T}(x)$ always lies below any secant line, and every secant line will have a slope less than three. ◀

Proof. (Theorem 11) We would like to bound the rendezvous time and to do this we will first bound the containment time. Since the robots must rendezvous by the second turning-point of $\mathcal{R}$ after the containment time, and since $\frac{|X_{k+2}|}{|X_k|} \leq 8$, the rendezvous time will be bounded by eight times the containment time. By Lemma 15 the containment time is itself bounded by three times the containment position and thus we will actually determine a bound on the containment position. Thus, if T_*, t_0, and x_0 are respectively the rendezvous time, containment time, and containment position, then $T_* < 8t_0 < 24|x_0|$. We will assume that $x_0 > 0$ and note that, since $\mathcal{T}(x)$ contains $\mathcal{T}'(x)$, we must have $d > 0$ in order for $x_0 > 0$.

To begin, we note that x_0 is the solution to the equation $\mathcal{T}'(x) - \mathcal{T}(x) = 0$, and, since this difference is increasing, any x satisfying $\mathcal{T}'(x) - \mathcal{T}(x) > 0$ will suffice for a bound. In particular, since $\mathcal{T}'(x) = \tau\mathcal{T}(\frac{x-d}{\tau}) > \tau\mathcal{T}(\frac{x}{\tau}) - 3d$, we have $\mathcal{T}'(x) - \mathcal{T}(x) > \tau\mathcal{T}(\frac{x}{\tau}) - \mathcal{T}(x) - 3d$ and we will thus bound x_0 by an x satisfying $\tau\mathcal{T}(\frac{x}{\tau}) - \mathcal{T}(x) > 3d$.

To simplify notation we introduce the variable $y = 4\ln(2)x$ and set $f(y) = \frac{1}{W(y)} - \frac{1}{W(\frac{y}{\tau})}$ and $D(y) = y \cdot f(y)$. We therefore wish to find a y satisfying $D(y) = y \cdot f(y) > 3d$. By (8) and (7) we can write $f(y) \geq \frac{1}{\ln(y) - \ln(\ln(y)) + \frac{e}{e-1}\frac{\ln(\ln(y))}{\ln(y)}} - \frac{1}{\ln(\frac{y}{\tau}) - \ln(\ln(\frac{y}{\tau})) + \frac{\ln(\ln(\frac{y}{\tau}))}{2\ln(\frac{y}{\tau})}}$. Now set $z = \ln(y)$. The right hand side of the above inequality then becomes

$$h(z) = \frac{1}{z - \ln(z) + \frac{e}{e-1}\frac{\ln(z)}{z}} - \frac{1}{z - \ln(\tau) - \ln(z - \ln(\tau)) + \frac{\ln(z - \ln(\tau))}{2(z - \ln(\tau))}}.$$

One can confirm that $h(z)$ admits a generalized Puiseaux series in the limit $z \to \infty$. Keeping only the leading term of this series we find that $h(z) = \frac{\ln(\frac{1}{\tau})}{z^2} + \mathcal{O}\left(\frac{\ln^2(z)}{z^3}\right)$. We can therefore conclude that $f(y) > \frac{\ln(\frac{1}{\tau})}{\ln^2(y)}$, $D(y) > \frac{\ln(\tau)y}{\ln^2(y)}$, and we now wish to find a y satisfying $\frac{\ln(\tau)y}{\ln^2(y)} \geq 3d$. Let y_+ be the solution to $\frac{\ln(\tau)y}{\ln^2(y)} = 3d$. We can use the Lambert W function to solve this equation. We find that $y_+ = \frac{12d}{\ln(\frac{1}{\tau})} W_{-1}^2\left(-\sqrt{\frac{\ln(\frac{1}{\tau})}{12d}}\right)$. Since the rendezvous time satisfies

$T_* < 8t_0 < 24x_0 < \frac{6y_+}{\ln(2)}$, we find that $T_* < \frac{72d}{\ln(2)\ln(\frac{1}{\tau})} W_{-1}^2 \left(-\sqrt{\frac{\ln(\frac{1}{\tau})}{12d}}\right)$. To express this in terms of more familiar functions we can use (9) to write

$$T_* < \frac{72d}{\ln(2)\ln(\frac{1}{\tau})} \ln^2\left(\sqrt{\frac{\ln(\frac{1}{\tau})}{12d}}\right) = \frac{18d\ln^2(d)}{\ln(2)\ln(\frac{1}{\tau})} + \mathcal{O}\left(\frac{d\ln(d)}{\ln(\frac{1}{\tau})}\right)$$

Expressing T_* with base-2 logarithms and dividing by $\frac{d}{1-\tau}$ gives the desired bound on the competitive ratio. ◀

If we abandon the use of algorithms with turning-points that are easily defined, then we can get an algorithm with a competitive ratio which is slightly "tunable". We claim the following:

▶ **Theorem 16.** *Rendezvous in V-Model is solved with a competitive ratio of* $\frac{72\ln^c(2)d\log(d)^{1+c}}{c|\log(\frac{1}{\tau})|}$ $+\mathcal{O}\left(\frac{d\log^c(d)}{c\log(\frac{1}{\tau})}\right)$ *using Algorithm 2 with* $\mathcal{T}(x) = 3|x| - \frac{|x|}{\ln^c(|x|)}$ *where $c > 0$ is a parameter of the algorithm.*

As the proof of Theorem 16 is essentially identical to that for Theorem 11 we do not provide it here.

There are a couple of things to note about this upper bound. First, although we can reduce the exponent of the $\log(d)$ term by making c small, we cannot make it arbitrarily small without suffering a large multiplicative constant due to the $\frac{1}{c}$ term in the competitive ratio. Thus, the algorithm of Theorem 16 will be most useful if one knows a lower bound on d as this will allow one to compare the bounds of Theorem 11 and 16 and choose an appropriate c. Without this knowledge it may be better to just stick with the algorithm of Theorem 11 as it has the benefit of having simple turning-points.

The upper-bounds of Theorem 11 and 16 are clearly much worse than the competitive ratios found for both the T- and D-Models. In those cases we had a constant competitive ratio and in these cases the competitive ratio is unbounded. One might then expect that we can do better. This, however, does not seem to be the case. We provide two arguments for this. First off, if one tries to use an algorithm in which the leading term of $\mathcal{T}(x)$ is $\omega(x)$ then we arrive to a similar result – the competitive ratio is unbounded.

▶ **Lemma 17.** *If the leading term of $\mathcal{T}(x)$ is $\omega(x)$ then the competitive ratio is $\omega(1)$.*

Proof. We observe that before the robots can rendezvous, there must be, at the very least, an intersection point (x_*, t_*) of the curves $\mathcal{T}(x)$ and $\mathcal{T}'(x)$ satisfying $0 < |x_*| < d$. We will show that we can always choose d and τ to make the time $t_* = \omega\left(\frac{d}{1-\tau}\right)$ if the leading term of $\mathcal{T}(x)$ is $\omega(x)$. For concreteness, assume that $\mathcal{R}'$ begins to the right of $\mathcal{R}$ such that x_* is the intersection point of the right arm of $\mathcal{T}(x)$ and the left arm of $\mathcal{T}'(x)$. Furthermore, since $\mathcal{T}(x) = \omega(x)$, we will express the leading order term of $\mathcal{T}(x)$ as $x \cdot f(x)$ for some positive function $f(x) = \omega(1)$.

We claim that x_* is bigger than $\frac{d}{2}$. Indeed, the right arm of $\mathcal{T}(x)$ is bounded from below by x and the left arm of $\mathcal{T}'(x)$ is bounded from below by $d - x$. Since $x = d - x$ when $x = \frac{d}{2}$, we must have $x_* > \frac{d}{2}$. However, if $x_* > \frac{d}{2}$ then $t_* > \mathcal{T}(\frac{d}{2}) > \frac{d}{2}f(\frac{d}{2})$. For any fixed $\tau \neq 1$ we can take d large enough that $\frac{d}{2}f(\frac{d}{2}) > \frac{d}{1-\tau}g(\frac{d}{1-\tau})$ for an appropriately chosen function $g(x) = \omega(1)$. Thus, $t_* = \omega\left(\frac{d}{1-\tau}\right)$ and, since the rendezvous time is larger than t_*, the lemma follows. ◀

Thus, the two algorithms analyzed in this section do seem to be from the "right" class of algorithms one should consider if one is to hope for a constant competitive ratio. The next lemma – which demonstrates why a geometric sequence of turning-points cannot be used – also supports this conclusion:

▶ **Lemma 18.** *If $\mathcal{T}(x)$ has the form $\mathcal{T}(x) = ax + G(x)$ with $a > 1$ and $|G(x)| = O(1)$ then there are choices of $\tau \neq 1$ such that the robots will never rendezvous.*

Proof. Let us first determine what the turning-points of $\mathcal{T}(x)$ look like. Since the turning-points satisfy the relation $\mathcal{T}(X_k) = T_k$ and $T_k = 2\sum_{i=0}^{k-1}|X_i| + |X_k|$ we have $a|X_k| + G(X_k) = 2\sum_{i=0}^{k-1}|X_i| + |X_k|$ and this simplifies to $|X_k| = \frac{2}{a-1}\sum_{i=0}^{k-1}|X_i| - G(X_k)$. Set $G_k = G(X_k)$ and observe that

$$\begin{aligned}|X_{k+1}| &= \frac{2}{a-1}\sum_{i=0}^{k}|X_i| - G_{k+1} = \frac{2}{a-1}|X_k| + \left(\frac{2}{a-1}\sum_{i=0}^{k-1}|X_i| - G_k\right) + G_k - G_{k+1}\\ &= \frac{a+1}{a-1}|X_k| + G_k - G_{k+1}\end{aligned}$$

and we can therefore see that X_{k+1} is nearly a geometric sequence with common ratio $\frac{a+1}{a-1}$.

Let $b = \frac{a+1}{a-1}$, and assume that $\tau = b^{-2}$ and that $\mathcal{R}'$ begins to the right of $\mathcal{R}$. In this case we can write

$$\begin{aligned}|X_{k+2}| &= b|X_{k+1}| + G_{k+1} - G_{k+2} = b(b|X_k| + G_k - G_{k+1}) + G_{k+1} - G_{k+2}\\ &= b^2|X_k| + bG_k - (b-1)G_{k+1} - G_{k+2}.\end{aligned}$$

Combining this with the fact that $|X'_{k+2}| = \tau|X_{k+2}| + d$ gives us

$$|X'_{k+2}| - |X_k| = d + \frac{1}{b^2}[bG_k - (b-1)G_{k+1} - G_{k+2}] = d + \Delta_k.$$

implying that the robot $\mathcal{R}'$ will be a distance $d + \Delta_k$ from its $(k+2)^{nd}$ turning-point when $\mathcal{R}$ reaches its kth turning-point. Clearly, in order to rendezvous, $d + \Delta_k$ must eventually decrease to zero. However, since $G(x) = \mathcal{O}(1)$, we can always take d sufficiently large such that this difference is bounded from below by a positive constant. Thus, with an appropriate choice of τ and d, the robots will never rendezvous. ◀

We are thus left with a rather small class of functions that $\mathcal{T}(x)$ can belong to – if $\mathcal{T}(x) = \omega(x)$ then the algorithm will take too much time, and if $\mathcal{T}(x) = a \cdot x \pm \mathcal{O}(1)$ then rendezvous cannot be solved. Thus, the only possibility left is if $\mathcal{T}(x) = a \cdot x + f(x)$ with $|f(x)| = o(x)$ and $|f(x)| = \omega(1)$. The two algorithms analyzed in this section each used curves of this form.

It is interesting to note that simulations of the two algorithms in this section show that there are choices of d and τ to (nearly) match the upper-bounds derived here. Even more interesting is that, in order to achieve these worst-case situations, one chooses d and τ precisely so that the two robots have turning-points that are arbitrarily close to the containment point of the curves $\mathcal{T}(x)$ and $\mathcal{T}'(x)$. This reflects a similar argument we made when we derived an upper-bound on rendezvous in the D-Model. In that case it also turned out that there were choices of d and τ in order to achieve the upper bound. If one could prove that it is always possible to choose d and τ such that the robots do have turning-points arbitrarily close to the containment point for all algorithms with $\mathcal{T}(x) = a \cdot x + f(x)$, $|f(x)| = o(x)$, and $|f(x)| = \omega(1)$ then a lower-bound that grows with d would easily follow. This is easier

said than done, however, and we leave it as an open problem whether or not one can achieve a constant competitive ratio in the V-Model.

Finally, we note that both algorithms provided in this section are universal in the sense that they will also solve the problem if both v and τ are different than one (it is trivial to see that one of $\mathcal{T}(x)$ or $\mathcal{T}'(x)$ will contain the other if $v \neq 1$). Since a robot does not need to know the values of its parameters in order to employ these algorithms we can conclude that it is sufficient to rendezvous if at least one of v or τ is different than one.

▶ **Theorem 19.** *Both of Algorithm 1 with $X_k = (k+2)2^k$ and Algorithm 2 with $\mathcal{T}(x) = 3|x| - \frac{|x|}{\ln^c(|x|)}$ solve rendezvous in general if at least one of v or τ is different than one.*

The time complexity of this more general model does not turn out to be all that interesting to study since, in the worst cases, an adversary chooses $v = 1$. Thus, the general model reduces to the V-Model and all of the results derived here still apply.

6 Discussion and conclusion

The focus of our paper was on symmetric rendezvous on an infinite line for two robots endowed with asymmetric clocks. After introducing the new concept of asymmetric clocks, we gave a universal algorithm which ensures feasibility of rendezvous if at least one of the robots' maximal speeds or time units differ. We analyzed the impact of equal time-unit, distance-unit, and equal speeds of the robots on the competitive ratio of the cost of rendezvous. The problem considered not only provides a surprising twist to the well-known rendezvous problem on an infinite line, it also creates interesting avenues for future research. These may include improving the algorithms, tightening bounds, employing robots that may have alternative capabilities (visibility and variable speed), as well as extensions to gathering for multiple robots.

References

1. S. Alpern and S. Gal. *The theory of search games and rendezvous*, volume 55. Kluwer Academic Publishers, 2002.
2. R. Baeza Yates, J. Culberson, and G. Rawlins. Searching in the plane. *Information and Computation*, 106(2):234–252, 1993.
3. R. Baeza-Yates and R. Schott. Parallel searching in the plane. *Computational Geometry*, 5(3):143–154, 1995.
4. L. Barrière, P. Flocchini, P. Fraigniaud, and N. Santoro. Rendezvous and Election of Mobile Agents: Impact of Sense of Direction. *Theory Comput. Syst.*, 40(2):143–162, 2007.
5. A. Beck. On the linear search problem. *Israel Journal of Mathematics*, 2(4):221–228, 1964.
6. A. Beck and D. Newman. Yet more on the linear search problem. *Israel Journal of Mathematics*, 8(4):419–429, 1970.
7. R. Bellman. An optimal search. *SIAM Review*, 5(3):274–274, 1963.
8. S. Bouchard, Y. Dieudonné, A. Pelc, and F. Petit. On deterministic rendezvous at a node of agents with arbitrary velocities. *Inf. Process. Lett.*, 133:39–43, 2018.
9. Q. Bramas and S. Tixeuil. Wait-Free Gathering Without Chirality. In *Structural Information and Communication Complexity - 22nd International Colloquium, SIROCCO 2015, Montserrat, Spain, July 14-16, 2015, Post-Proceedings*, pages 313–327, 2015.
10. M. Chrobak, L. Gasieniec, Gorry T., and R. Martin. Group Search on the Line. In *Proceedings of SOFSEM 2015, LNCS 8939*, pages 164–176. Springer, 2015.

11 R. Cohen and D. Peleg. Convergence of autonomous mobile robots with inaccurate sensors and movements. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 549–560. Springer, 2006.

12 A. Collins, J. Czyzowicz, L. Gasieniec, and A. Labourel. Tell me where I am so I can meet you sooner. In *International Colloquium on Automata, Languages, and Programming*, pages 502–514. Springer, 2010.

13 R. M. Corless, G. H. Gonnet, D. E. G. Hare, D. J. Jeffrey, and D. E. Knuth. On the Lambert W function. *Advances in Computational Mathematics*, 5(1):329–359, December 1996.

14 J. Czyzowicz, E. Kranakis, D. Krizanc, L. Narayanan, and J. Opatrny. Search on a Line with Faulty Robots. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC*, pages 405–414, 2016.

15 J. Czyzowicz, E. Kranakis, D. Krizanc, L. Narayanan, J. Opatrny, and S. Shende. Linear search with terrain-dependent speeds. In *International Conference on Algorithms and Complexity*, pages 430–441. Springer, 2017.

16 G. De Marco, L. Gargano, E. Kranakis, D Krizanc, A. Pelc, and U. Vaccaro. Asynchronous deterministic rendezvous in graphs. *Theoretical Computer Science*, 355(3):315–326, 2006.

17 E. D. Demaine, S. P. Fekete, and S. Gal. Online searching with turn cost. *Theoretical Computer Science*, 361(2):342–355, 2006.

18 A. Dessmark, P. Fraigniaud, D. R. Kowalski, and A. Pelc. Deterministic Rendezvous in Graphs. *Algorithmica*, 46(1):69–96, 2006.

19 Y. Dieudonné, A. Pelc, and V. Villain. How to Meet Asynchronously at Polynomial Cost. *SIAM J. Comput.*, 44(3):844–867, 2015.

20 O. Feinerman, A. Korman, S. Kutten, and Y. Rodeh. Fast rendezvous on a cycle by agents with different speeds. *Theoretical Computer Science*, 688:77–85, 2017.

21 A. Hoorfar and M. Hassani. Inequalities on the Lambert W function and hyperpower function. *JIPAM. Journal of Inequalities in Pure & Applied Mathematics [electronic only]*, 9, January 2008.

22 T. Izumi, S. Souissi, Y. Katayama, N. Inuzuka, X. Défago, K. Wada, and M. Yamashita. The Gathering Problem for Two Oblivious Robots with Unreliable Compasses. *SIAM J. Comput.*, 41(1):26–46, 2012.

23 M.-Y. Kao, J. H. Reif, and S. R. Tate. Searching in an unknown environment: An optimal randomized algorithm for the cow-path problem. *Information and Computation*, 131(1):63–79, 1996.

24 E. Kranakis, N. Santoro, C. Sawchuk, and D. Krizanc. Mobile agent rendezvous in a ring. In *Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on*, pages 592–599. IEEE, 2003.

25 A. Pelc. Deterministic rendezvous in networks: A comprehensive survey. *Networks*, 59(3):331–347, 2012.

26 G. Stachowiak. Asynchronous Deterministic Rendezvous on the Line. In *SOFSEM 2009: Theory and Practice of Computer Science*, pages 497–508. Springer, 2009.

27 X. Yu and M. Yung. Agent rendezvous: A dynamic symmetry-breaking problem. In *International Colloquium on Automata, Languages, and Programming*, pages 610–621. Springer, 1996.

Approximate Neighbor Counting in Radio Networks

Calvin Newport[1]
Georgetown University, Washington, D.C., United States
cnewport@cs.georgetown.edu

Chaodong Zheng[2]
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
chaodong@nju.edu.cn

Abstract

For many distributed algorithms, neighborhood size is an important parameter. In radio networks, however, obtaining this information can be difficult due to ad hoc deployments and communication that occurs on a collision-prone shared channel. This paper conducts a comprehensive survey of the approximate neighbor counting problem, which requires nodes to obtain a constant factor approximation of the size of their network neighborhood. We produce new lower and upper bounds for three main variations of this problem in the radio network model: (a) the network is single-hop and every node must obtain an estimate of its neighborhood size; (b) the network is multi-hop and only a designated node must obtain an estimate of its neighborhood size; and (c) the network is multi-hop and every node must obtain an estimate of its neighborhood size. In studying these problem variations, we consider solutions with and without collision detection, and with both constant and high success probability. Some of our results are extensions of existing strategies, while others require technical innovations. We argue this collection of results provides insight into the nature of this well-motivated problem (including how it differs from related symmetry breaking tasks in radio networks), and provides a useful toolbox for algorithm designers tackling higher level problems that might benefit from neighborhood size estimates.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Radio networks, neighborhood size estimation, approximate counting

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.26

Related Version A full version of the paper is available at [20], https://arxiv.org/abs/1811.03278.

1 Introduction

Many distributed algorithms assume nodes have advance knowledge of their neighborhood, allowing them to take steps that depend, for example, on gathering information from every neighbor (e.g., [16]), or flipping a coin weighted with their neighborhood size (e.g., [1]).

In standard wired network models, obtaining this neighbor information is often trivial (e.g., as in the LOCAL or CONGEST models). In radio networks, by contrast, this information might be harder to obtain. Specifically, because nodes in these networks are often deployed in an ad hoc manner, and subsequently communicate only on a contended shared channel, we

[1] Supported by NSF award 7773087.
[2] Supported by National Key R&D Program of China 2018YFB1003200, and NSFC 61702255.

© Calvin Newport and Chaodong Zheng;
licensed under Creative Commons License CC-BY
22nd International Conference on Principles of Distributed Systems (OPODIS 2018).
Editors: Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira; Article No. 26; pp. 26:1–26:16

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

cannot expect that they possess advance knowledge of their neighborhood. In fact, learning this information might require non-trivial feats of contention management.

Some distributed algorithms for radio networks depend on nodes possessing an estimate of their neighborhood size (e.g., [9, 10]), while other algorithms could be significantly simplified if this information was available (e.g., [22, 18, 13]). Though it is generally assumed that calculating these size estimates should not take *too* long in most settings, this problem has escaped the more systematic scrutiny applied to related tasks like contention resolution.

In this paper, we work toward filling in more of this knowledge gap. We conduct a comprehensive survey of lower and upper bounds for the approximate neighbor counting problem in the radio network model under different combinations of common assumptions for this setting. Some of our results require only extensions of existing strategies, while many others require non-trivial technical innovations.

Combined, this collection of results provides two important contributions to the study of distributed algorithms for radio networks. First, it supports a deeper understanding of the well-motivated neighbor counting problem, highlighting both its similarities and differences to related low-level radio network tasks. Second, the collection acts as a useful toolbox for algorithm designers tackling higher level problems.

Result summary. The radio network model we study describes the underlying network topology with an undirected connected graph $G = (V, E)$, with the $n = |V|$ vertices corresponding to the radio devices (usually called *nodes* in this paper), and the edges in E describing which node pairs are within communication range. For every node $u \in V$, n_u describes the number of neighbors of u in G. We sometimes call this parameter the *neighbor count* of u. In single-hop networks (i.e., G is a clique), all nodes have the same neighbor count, while in multi-hop networks these counts can differ.

The *approximate neighbor counting* problem requires nodes to calculate constant factor estimates of their neighbor counts. We study the variant where every node must obtain this estimate (e.g., during network initialization), and the variant where only a designated node must obtain this estimate (e.g., when neighborhood of a node changes). We study these variants in single-hop and multi-hop networks, and consider solutions with and without collision detection. We study both lower and upper bounds for randomized solutions. When relevant, we look at both results that hold with constant and high probability.

Our results are summarized in Table 1. Notice that we do not study both designated and all nodes counting in single-hop networks, as in this setting all nodes have the same neighbor count, making these two cases essentially identical (e.g., a designated node in a single-hop network can simply announce its count, transforming the solution to an all nodes counting solution). We also do not study constant probability solutions for all nodes counting in multi-hop networks. This follows because in the multi-hop setting the success probability applies to each individual node. A constant success probability, therefore, implies that a constant fraction of the nodes are expected to generate inaccurate neighbor counts – a result that is too weak in most scenarios. In the single-hop setting, by contrast, the success probability refers to the probability that *all* nodes generate good counts.

Also notice that two upper bounds are given for multi-hop all nodes counting without collision detection: $O(\lg^2 n_u)$ and $O(\lg^3 N)$. The first bound describes an algorithm that generates good neighbor counts but never terminates (specifically, each node must keep participating to help neighbors that are still counting). The second bound does terminate, but requires an upper bound N on the maximum possible network size. This is the only algorithm we study that requires this information to work properly.

Table 1 Summary of all approximate neighbor counting results proved in this paper. In the above table, "CD" and "no-CD" denote "with collision detection" and "without collision detection" respectively, while "high probability" is expressed with respect to the parameter in the bound. The N and N_Δ terms describe upper bounds on the maximum neighbor count in single-hop and multi-hop networks, respectively. Our lower bounds are expressed with respect to these maximum sizes, while our upper bounds are expressed with respect to the actual network sizes, with the exception of the $O(\lg^3 N)$ bound for multi-hop all nodes counting without collision detection. This is the only algorithm we study that requires knowledge of network statistics to work properly.

		with constant probability		with high probability	
		no-CD	CD	no-CD	CD
all nodes in single-hop	lower bound	$\Omega(\lg N)$	$\Omega(\lg\lg N)$	$\Omega(\lg^2 N)$	$\Omega(\lg N)$
	upper bound	$O(\lg n)$	$O(\lg\lg n)$	$O(\lg^2 n)$	$O(\lg n)$
designated node in multi-hop	lower bound	$\Omega(\lg N_\Delta)$	$\Omega(\lg\lg N_\Delta)$	$\Omega(\lg^2 N_\Delta)$	$\Omega(\lg N_\Delta)$
	upper bound	$O(\lg n_w)$	$O(\lg\lg n_w)$	$O(\lg^2 n_w)$	$O(\lg n_w)$
all nodes in multi-hop	lower bound	–	–	$\Omega(\lg^2 N_\Delta)$	$\Omega(\lg N_\Delta)$
	upper bound	–	–	$O(\lg^2 n_u)$, $O(\lg^3 N)$	$O(\lg^2 n_u)$

Discussion. For all but one cases that we have lower bounds, they match our upper bounds. For the single-hop results, these bounds also match the relevant bounds from the related single-hop contention resolution problem (c.f., [19]). In fact, most of the lower bounds in this single-hop setting follow by reduction from contention resolution. That is, we show that if you can solve approximate neighbor counting fast, then you can also solve contention resolution fast – allowing existing lower bounds from the latter to carry over to the former.

The single-hop upper bounds, however, required more than the simple application of existing contention resolution strategies. In contention resolution, for example, if you get lucky with your coin flips, and a node broadcasts alone earlier than expected, this is good news – you have solved the problem even faster! In neighbor counting, however, this "luck" might lead you to output an inaccurate size estimate. The analysis used for neighbor counting must bound the probabilities of these precocious symmetry breaking events.

Another complexity of neighbor counting (in single-hop networks) as compared to contention resolution is that *all* nodes must learn an estimate. This requires extra mechanisms to ensure that once some nodes learn a good estimate, this information is spread to all others. The most difficult single-hop case is the combination of high probability correctness and collision detection. To achieve an accurate estimate in an optimal $O(\lg n)$ rounds required the adaptation of a technique based on one-dimensional random walks [18, 3].

Obtaining lower bounds for the multi-hop designated node setting required technical innovations. In the single-hop setting, our lower bounds used reduction arguments that applied the contention resolution bounds from [19] as a black box. In the multi-hop designated node setting, by contrast, we were forced to open the black boxes and modify them to handle the issues specific to multi-hop topologies. For the particular case of collision detection and high probability, substantial new arguments were needed to transform the bound.

In the multi-hop all nodes setting, obtaining upper bounds also required techniques beyond standard symmetry breaking strategies, as each node may simultaneously participate in multiple estimation processes. Our collision detector algorithm for this case has nodes use detectable noise to notify neighbors that they are still counting. When collision detection

is not available, we consider two different approaches and hence present two algorithms. The first one returns an estimate for n_u in $O(\lg^2 n_u)$ rounds, which is correct with high probability in n_u. The second algorithm uses a "double counting" trick and takes longer time, but the returned estimate is correct with high probability in N.

Last but not least, we would like to clarify a point about our lower bound statements. As shown in Table 1, our lower bounds are expressed with respect to the maximum possible neighbor counts (e.g., N and N_Δ), whereas, to obtain the strongest possible results, our upper bounds are expressed with respect to the actual neighbor counts in the analyzed execution (e.g., n and n_u). The right way to interpret our lower bounds is that they claim in a setting where the number of participants comes from a set of N (or N_Δ) possible participants, there *exists* a subset of these participants for which the stated bound holds.

Our lower bound technique does not directly tell us anything about the *size* of the participant set that induces the slow performance. Given our matching upper bounds, however, we can conclude that the worst case participant sets for these algorithms must have a size close to the maximum bounds. Consider, for example, single-hop counting with no collision detection. The lower bound says that for each algorithm there exists a collection of no more than N participants that requires $\Omega(\lg N)$ rounds to generate a good count with constant probability. Our upper bound, on the other hand, guarantees a good count in $O(\lg n)$ rounds with constant probability, where n is the size of the participant set. It follows that when the lower bound is applied to our algorithm, the bad participant set must have a size that is polynomial in N (i.e., $n = \Theta(N^\gamma)$, for some constant $0 < \gamma \leq 1$), as otherwise the existence of both bounds is a logical contradiction.

2 Related Work

Algorithms to reduce contention and enable communication on shared channels date back to the early days of networking (c.f., [11, 6]), and remain an active area of study today. In the study of distributed algorithms for shared *radio* channels, many strategies explicitly execute approximate neighbor counting as a subroutine. For example, in their study of energy-efficient initialization with collision detection, Bordim et al. [2] propose a protocol that returns an estimate of n in the range $[n/(16\lg n), 2n/\lg n]$ within $O(\lg^2 n)$ time, while requiring each node to be awake for at most $O(\lg n)$ rounds. Similarly, Gilbert et al. [10] use approximate neighbor counting as part of a neighbor discovery protocol in cognitive radio networks. It is also common for algorithms in this setting to simply assume these estimates are provided in advance. E.g., the often-used *decay* strategy introduced by Bar-Yehuda et al. [1], requires a bound on local neighborhood size to limit the estimates it tests.

As mentioned throughout this paper, neighbor counting is often closely related to contention resolution, which requires a single node to broadcast alone on the channel. Some common contention resolution strategies implicitly provide this approximation as a side-effect of their operation (e.g., [22, 18, 13]). At the same time, under some assumptions, a good estimate simplifies the problem of contention resolution. As we detail throughout this paper, however, this relationship is not exact. Lower bounds for neighbor counting often require more intricate arguments than contention resolution, and in some cases, contention resolution algorithms require nontrivial extra analysis and mechanisms to provide counts. Teasing apart this intertwined relationship is one of the main contributions of this paper.

Others have directly studied approximate neighbor counting in radio networks. Jurdzinski et al. [12] develop an algorithm that provides a constant factor approximation of n within $O(\lg^{2+\delta} n)$ time without collision detection for arbitrary constant $\delta > 0$. Their algorithm

guarantees that no node participates in more than $O((\lg\lg n)^{\delta})$ rounds. (Our relevant algorithm only needs $O(\lg^2 n)$ rounds, but consumes more energy.) Caragiannis et al. [4] devise two constant-factor approximation algorithms: the first one requires collision detection and takes $O((\lg n)\cdot(\lg\lg n))$ time, while the second one works without collision detection and takes $O(\lg^2 n)$ time. (Our relevant algorithms only need $O(\lg n)$ rounds with collision detection, and perform as well as theirs without collision detection.) In [14, 15], the authors discuss how to approximate network size when adversaries are present.

Approximate neighbor counting has also been studied in the BEEPING model [7], which is similar to, but somewhat weaker than, the standard radio network model. In this setting, Chen et al. [5] conduct an excellent mini survey on recent works in RFID counting (e.g., [24, 21, 23, 5]). They conclude that a two-phase approach is the key to achieve efficient and accurate RFID counting. They also prove several lower bounds, one of which shows $\Omega(\lg\lg n)$ rounds are needed to obtain a constant factor approximation with constant probability. More recently, Brandes et al. [3] study how to efficiently estimate the size of a single-hop BEEPING network: they provide both lower and upper bounds for a parameterized approximation accuracy. Notice, the main objective of [5] and [3] differs from ours not just in the model, but in that they seek a $(1+\epsilon)$ approximation of n for any $\epsilon > 0$ (ϵ can be non-constant). Nonetheless, they both use constant factor approximation as a key subroutine.

3 Model and Problem

We consider a synchronous radio network. We model the topology of this network with a connected undirected graph $G = (V, E)$, with the $n = |V|$ vertices corresponding to the radio devices (usually called *nodes* in this paper), and the edges in E describing which node pairs are within communication range.

For each node $u \in V$, we use Γ_u to denote the set of neighbors of u, and use $n_u = |\Gamma_u|$ to denote the number of neighbors of u. Let $n_{\Delta} = \max_{u\in V}\{n_u\}$. Our algorithms assume $n_u \geq 1$. That is, we do not confront the possibility of a node isolated from the rest of a multi-hop network, or a single-hop network consisting of only a single node (we see the so-called *loneliness detection* problem as an interesting but somewhat orthogonal challenge; e.g., [8]). For the ease of presentation, we assume n_u and n are always a power of two. This assumption does not affect the correctness or asymptotic time complexities of our results. We define N and N_{Δ} to be upper bounds on the maximum possible size of n and n_{Δ}, respectively. To obtain the strongest and most general possible results, our algorithms are *not* provided with knowledge of N and N_{Δ}, with the exception of an $O(\lg^3 N)$ time algorithm for multi-hop all nodes counting without collision detection.

We divide time into discrete and synchronous *slots* that we also sometimes call *rounds*. We assume all nodes start execution during the same slot. (The definition of "neighbor counting" becomes complicated once nodes can activate in different time slots.) These assumptions imply nodes have access to a global clock. We assume each node is equipped with a half-duplex radio transceiver. That is, in each time slot, each node can choose to broadcast or listen, but cannot do both. If a node chooses to broadcast, then it gets no feedback from the communication channel. If a node chooses to listen and no neighbors of it broadcasts, then the node hears nothing (i.e., silence). If a node chooses to listen and exactly one of its neighbors broadcasts, then the node receives the message from that neighbor. Finally, if a node chooses to listen and at least two of its neighbors broadcast, then the result depends on the availability of a *collision detection* mechanism: if collision detection is available, then the listening node hears noise; otherwise, the listening node hears nothing.

As a result, without collision detection, a listening node cannot tell whether there are no neighbors broadcasting or there are multiple neighbors broadcasting.

In this paper, we are interested in the *approximate neighbor counting* problem. This problem requires selected node(s) to obtain a constant factor approximation of their neighborhood size(s). In more detail, let constant $\tilde{c} \geq 1$ be the fixed approximation threshold for this problem. Each node u that produces an estimate $\hat{n}_u$ must satisfy $n_u \leq \hat{n}_u \leq \tilde{c} \cdot n_u$. We consider three variations of this problem that differ with respect to the allowable network topologies and requirements on which nodes produce an estimate. The first variant assumes G is single-hop and all nodes must produce an *identical* estimate. The second variant assumes G is multi-hop, but only a *single* designated node w must produce an estimate. The third variant is the same as the second, except that now *every* node must produce an estimate. We study randomized algorithms that are proved to be correct with a given probability p. In the single-hop variant, p describes the probability of the event in which all nodes generate a single good approximation. In the multi-hop variants, by contrast, p is the probability that an individual counting node generates a good approximation.

Throughout this paper, we cite the related *contention resolution* problem. In single-hop networks, the contention resolution problem is solved once some node broadcasts alone. Later in the paper, we consider a version of multi-hop contention resolution in which a single designated node must receive a message from a neighbor to solve the problem.

Finally, in the following, we say an event occurs *with high probability in parameter k* (or "w.h.p. in k") if it occurs with probability at least $1 - 1/k^{\gamma}$, for some constant $\gamma \geq 1$.

4 Lower Bounds

In this section, we presents our lower bounds for the approximate neighbor counting problem. We begin, in Section 4.1 by looking at lower bounds that can be proved by reducing from the contention resolution problem. That is, in that subsection, we prove lower bounds by arguing that solving neighbor counting fast implies an efficient algorithm to contention resolution, allowing the relevant contention resolution lower bounds to apply.

We employ this approach to derive bounds for constant probability and high probability counting with no collision detection in both single-hop and designated node multi-hop settings. We also apply this approach to derive bounds for constant probability counting with collision detection in these settings. We cannot, however, apply this approach to high probability counting with collision detection, as the reduction itself is too slow compared to the desired bounds. We note that for the single-hop arguments, we leverage existing contention resolution bounds from [19]. For the multi-hop arguments, however, we must first generalize the results from [19] to hold for the considered network topology.

In Section 4.2, we look at lower bounds for high probability approximate neighbor counting with collision detection in both single-hop and designated node multi-hop settings. Unlike in Section 4.1, we cannot deploy a reduction-based argument. We instead prove a new lower bound that directly argues a sufficiently accurate estimate requires the stated rounds.

Finally, in Section 4.3 we look at lower bounds for the remaining case of multi-hop all nodes counting. We establish these bounds by reduction from designated node multi-hop bounds, as solving all nodes counting trivially also solves designated node counting.

4.1 Lower Bounds via Reduction from Contention Resolution

We begin with our lower bound arguments that rely on reductions from contention resolution. For the single-hop scenario, we can reduce from single-hop contention resolution and apply existing lower bounds from [19]. (See full version of the paper [20] for details on contention

resolution lower bounds in single-hop networks.) For multi-hop designated node counting, however, we must first prove new contention resolution lower bounds.

In particular, consider the definition of multi-hop contention resolution in which there is a well-defined designated node w, and the goal is for exactly one of w's neighbors – which is a size n_w subset drawn from a size N_Δ universe – to broadcast alone in some time slot. At first glance, this problem might seem easier than single-hop contention resolution as we are provided with a designated node w that could coordinate its neighbors in their quest to break symmetry among themselves. We prove, however, that this is not the case: the lower bounds are the same as their single-hop counterparts. In more detail, we prove the following two lemmas by adapting the techniques from [19] to this new set of assumptions (see full version of the paper [20] for the omitted proofs of this section):

▶ **Lemma 1.** *Let $\mathcal{A}$ be an algorithm that solves contention resolution in $g(N_\Delta)$ time slots with probability p in multi-hop networks with no collision detection. It follows that: (a) if p is some constant, then $g(N_\Delta) \in \Omega(\lg N_\Delta)$; and (b) if $p \geq 1 - 1/N_\Delta$, then $g(N_\Delta) \in \Omega(\lg^2 N_\Delta)$.*

▶ **Lemma 2.** *Let $\mathcal{A}$ be an algorithm that solves contention resolution in $g(N_\Delta)$ time slots with probability p in multi-hop networks with collision detection. It follows that if p is some constant, then $g(N_\Delta) \in \Omega(\lg\lg N_\Delta)$.*

With the needed contention resolution lower bounds in hand, we turn our attention to reducing this problem to approximate neighbor counting. Take the single-hop scenario as an example, the basic idea behind the reduction is that once nodes have an estimate $\hat{n}$ of n, they can simply broadcast with probability $1/\hat{n}$ in each time slot. If this estimate is good, then in each time slot, they have a constant probability of isolating a broadcaster, thus solving contention resolution. Moreover, repeating this step multiple times increases the chance of success proportionally. Building on these basic observations, we prove the following:

▶ **Lemma 3.** *Assume there exists an algorithm $\mathcal{A}$ that solves approximate neighbor counting in $h(N)$ (or, $h(N_\Delta)$ in the multi-hop scenario) time slots with probability p. Then, there exists an algorithm $\mathcal{B}$ that solves contention resolution in $2(h(N)+k)$ (resp., $2h(N_\Delta)+k$ in the multi-hop scenario) time slots with probability at least $(1-e^{-k/(4\tilde{c})}) \cdot p$. Here, $k \geq 1$ is an integer, and $\tilde{c} \geq 1$ is the constant defined in Section 3.*

Combining the reduction described in Lemma 3 with the single-hop lower bounds for contention resolution from [19] and the new multi-hop lower bounds proved above, we get the following lower bounds for approximate neighbor counting:

▶ **Theorem 4.** *In a single-hop radio network containing at most N nodes:*

- *When collision detection is not available, solving approximate neighbor counting with constant probability requires $\Omega(\lg N)$ time in the worst case; solving approximate neighbor counting with high probability in N requires $\Omega(\lg^2 N)$ time in the worst case.*
- *When collision detection is available, solving approximate neighbor counting with constant probability requires $\Omega(\lg\lg N)$ time in the worst case.*

In a multi-hop radio network in which the designated node has at most N_Δ neighbors:

- *When collision detection is not available, solving approximate neighbor counting with constant probability requires $\Omega(\lg N_\Delta)$ time in the worst case; solving approximate neighbor counting with high probability in N_Δ requires $\Omega(\lg^2 N_\Delta)$ time in the worst case.*
- *When collision detection is available, solving approximate neighbor counting with constant probability requires $\Omega(\lg\lg N_\Delta)$ time in the worst case.*

4.2 Custom Lower Bounds for High Probability and Collision Detection

At this point, for single-hop and designated node multi-hop variants of the approximate neighbor counting problem, the only lower bounds missing are the ones of ensuring high success probability with collision detection. As we detail in full version of the paper [20], our previous reduction-based approach no longer works in these scenarios. (Roughly speaking, the reduction itself takes at least as long as the lower bound we intend to prove.) Therefore, we must construct custom lower bounds for this problem and exact set of assumptions.

We start by proving the following combinatorial result:

▶ **Lemma 5.** *Let c and k be two positive integers such that $c \leq k$. Let $\mathcal{R}$ be the set containing all size c subsets from $[k] = \{1, 2, \cdots, k\}$. Let $\mathcal{H}$ be an arbitrary set of size less than $\lg(k/c)$ such that each element in $\mathcal{H}$ is a subset of $[k]$. Then, there exists some $R \in \mathcal{R}$ such that for each $H \in \mathcal{H}$, either $R \subseteq H$ or $R \cap H = \emptyset$.*

Intuitively, a set $\mathcal{H}$ can be interpreted as a *broadcast schedule* generated by an algorithm $\mathcal{A}$: a node labeled i broadcasts in slot j if and only if it is activated, and i is in the jth set in $\mathcal{H}$. Given this interpretation, Lemma 5 suggests: for both the single-hop and the multi-hop designated node scenario, for any approximate neighbor counting algorithm $\mathcal{A}$, and for any broadcast schedule generated by $\mathcal{A}$ of length less than $\lg(k/c)$, there exists a set of c nodes (or a set of c neighbors of the designated node in the multi-hop scenario) such that if these c nodes are activated and execute $\mathcal{A}$, then during each of the first $\lg(k/c) - 1$ time slots, either none of them broadcast or all of them broadcast. This further implies, if only two of these c nodes are activated, then their view (of the first $\lg(k/c) - 1$ time slots of the execution) is indistinguishable from the case in which all of these c nodes are activated.

Now, imagine an adversary who samples a size c subset from $\mathcal{R}$ with uniform randomness, and then flips a fair coin to decide whether to activate all these c nodes, or just two of them. If the adversary happens to have chosen the set R proved to exist in Lemma 5, then by the end of slot $\lg(k/c) - 1$, algorithm $\mathcal{A}$ cannot distinguish between two and c nodes. Notice, if c is large compared to the approximation threshold $\tilde{c}$, then this difference matters: outputting two when the real count is c (or vice versa) is unacceptable. Thus, in such case, the algorithm gets the right answer with only probability $1/2$ – not enough for high success probability.

A complete and rigorous proof for the above intuition is actually quite involved, again see full paper [20] for more details. In the end, we obtain the following lower bounds:

▶ **Theorem 6.** *Assume collision detection is available, then:*

- *In a single-hop radio network containing at most N nodes, solving approximate neighbor counting with high probability in N requires $\Omega(\lg N)$ time in the worst case.*
- *In a multi-hop radio network in which the designated node has at most $N_\triangle$ neighbors, solving approximate neighbor counting with high probability in $N_\triangle$ requires $\Omega(\lg N_\triangle)$ time in the worst case.*

4.3 All Nodes Multi-Hop Lower Bounds

If an algorithm can solve multi-hop all nodes approximate neighbor counting, then clearly the same algorithm can be used to solve multi-hop designated node approximate neighbor counting, with same time complexity and success probability. Therefore, the lower bounds we previously proved for the latter variant naturally carries over to the former variant:

▶ **Theorem 7.** *In a multi-hop radio network containing at most N nodes:*

- *When collision detection is not available, solving approximate neighbor counting with high probability in N requires $\Omega(\lg^2 N)$ time in the worst case.*
- *When collision detection is available, solving approximate neighbor counting with high probability in N requires $\Omega(\lg N)$ time in the worst case.*

5 Upper Bounds

In this section, we describe and analyze several randomized algorithms that solve the approximate neighbor counting problem. We will begin with single-hop all nodes counting. Specifically, four algorithm are presented for this variant, each based on a different approach. Though most of the strategies used are previously known, extensions to design and analysis are often needed. We then introduce three algorithms for multi-hop all nodes counting, including one which is particularly interesting, as it uses a "double counting" trick that is not related to contention resolution at all to obtain high success probability. Finally, we briefly discuss solutions for multi-hop designated node counting, as most of these algorithm are simple variations of their counterparts for single-hop all nodes counting.

Due to space constraint, if not otherwise stated, complete proofs for lemma and theorem statements are provided in the appendix. Nonetheless, we will usually discuss the intuitions or high-level strategies for proving them.

5.1 Single-Hop Networks: No Collision Detection

Our algorithms often adopt a classical technique inspired by the contention resolution literature: "*guess and verify*". In more detail, take a *guess* about the count, and then *verify* its accuracy; if the guess is good enough then we are done, otherwise take another guess and repeat. This simple approach is versatile: depending on how the guesses are made and verified, many variations exist, resulting in efficient algorithms suitable for different settings.

A standard approach to this guessing is to use a geometric sequence with common ratio two, which is usually called (exponential) *decay* [1]. This sequence leverages the fact that we only need a constant factor estimate to speed things up. Particularly, if the real count is n, then only $O(\lg n)$ iterations are needed before reaching an accurate estimate.

Once a guess is made, we need to verify its accuracy. To accomplish this, it is sufficient to let each participating node broadcast with a probability proportional to the reciprocal of the guess, and then observe the status of the channel. The intuition is simple: underestimate will result in collision and overestimate will result in silence; and we expect one node to broadcast alone – a distinguishable event – iff the estimate is accurate enough. The algorithms described below adapt this general approach to their specific constraints.

Constant probability of success. We now present COUNT-SH-noCD-CONST. (In the algorithm's name, SH means "single-hop", noCD means "no collision detection", and CONST means "success with constant probability".) This algorithm applies the most basic form of the "guess and verify" strategy. It provides a correct estimate with constant probability in $O(\lg n)$ time for single-hop radio networks, when collision detection is not available.

COUNT-SH-noCD-CONST contains multiple iterations, each of which has two time slots. In the ith iteration, nodes assume $n \approx 2^i$, and verify whether this estimate is accurate or not. More specifically, in the first time slot within the ith iteration, each node will broadcast a beacon message with probability $1/2^i$ and listen otherwise. If a node decides to listen and hears a beacon message in the first time slot, then it will set its estimate to 2^{i+2} and terminate after this iteration. That is, if a single node u broadcasts alone in the first time

slot of iteration i, then all listening nodes – which is all nodes except u – will terminate by the end of this iteration, with 2^{i+2} being their estimate. Notice, we still need to inform u about this estimate, which is the very purpose of the second time slot within each iteration. More specifically, in the second time slot within the ith iteration, for each node u, if it has heard a beacon message in the first time slot of this iteration, then it will broadcast a stop message with probability $1/(2^i - 1)$. Otherwise, if u has broadcast in the first time slot of this iteration, then it will listen in this second time slot. Moreover, it will terminate with its estimate set to 2^{i+2}, if it hears a stop message in this second time slot.

Despite its simplicity, proving the correctness of COUNT-SH-NOCD-CONST requires efforts beyond what would suffice for basic contention resolution. First, by carefully calculating and summing up the failure probabilities, we show no node will terminate during the first $\lg n - 3$ iterations, with at least constant probability. Then, we show during iteration i where $\lg n - 2 \leq i \leq \lg n$, either no node terminates or all nodes terminate, with at least constant probability. Finally, we prove that if all nodes are still active by iteration $\lg n$, then all of them will terminate by the end of it, again with at least constant probability. Complete analysis can be found in the full paper [20], here we present only the main theorem:

▶ **Theorem 8.** *The* COUNT-SH-NOCD-CONST *approximate neighbor counting algorithm ensures the following properties with constant probability when executed in a single-hop network with no collision detection: (a) all nodes terminate simultaneously within $O(\lg n)$ slots; and (b) all nodes obtain the same estimate of n, which is in the range $[n, 4n]$.*

High probability of success. Observe that in the aforementioned simplest form of "guess and verify", as the estimate increases, the probability that multiple nodes broadcast decreases, and the probability that no node broadcasts increases. A more interesting metric is the probability that a single node broadcasts alone: it first increases, and then decreases; not surprisingly, the peak value is reached when the estimate is the real count. These facts suggest, for each estimate $\hat{n}$, we could repeat the procedure of broadcasting with probability $1/\hat{n}$ multiple times, and use the *fraction* of noisy/silent/clear-message slots to determine the accuracy of the estimate. This method can provide stronger correctness guarantees, but complicates *termination detection* (i.e., when should a node stop), as different nodes may observe different fraction values. For example, if some nodes have already obtained a correct estimate but terminate too early, then remaining nodes might never get correct estimates, since there are fewer nodes remaining. The situation becomes more challenging when an upper bound of the real count is not available. To resolve this issue, sometimes, we have to carefully craft and embed a "consensus" mechanism.

COUNT-SH-NOCD-HIGH highlights our above discussion. This algorithm contains multiple iterations, each of which has three phases. In the ith iteration, nodes assume $n \approx 2^i$. The first phase of each iteration i – which contains $\Theta(i)$ time slots – is used to verify the accuracy of the current estimate. In particular, in each slot within the first phase, each node will broadcast a beacon message with probability $1/2^i$, and listen otherwise. By the end of the first phase, each node will calculate the fraction of time slots (among all its listening slots in this phase) in which it has heard a beacon message. For each node, if at the end of the first phase of some iteration j, this fraction value has reached $1/2e$ for the first time since the start of execution, the node will set 2^j as its *private estimate* for n. Recall nodes might not obtain private estimates simultaneously, thus they cannot simply terminate and output private estimates as the final estimate. This is the place where the latter two phases come into play. More specifically, in the second phase, nodes that have already obtained private estimates will try to broadcast informed messages to signal other nodes to stop. In fact,

hearing an informed message is the only situation in which a node can safely terminate, even if the node has already obtained its private estimate. On the other hand, the third phase is used to deal with the case in which one single "unlucky" node successfully broadcasts an informed message during phase two (thus terminate all other nodes), but never gets the chance to successfully receive an informed message (thus cannot terminate along with other nodes). Complete description of Count-SH-noCD-High can be found in full paper [20].

To prove the correctness of Count-SH-noCD-High, we need to show: (a) nodes can correctly determine the accuracy of their estimates; and (b) all nodes terminate simultaneously and output identical estimate. Part (b) follows from our careful protocol design, as phase two and three in each iteration act like a mini "consensus" protocol, allowing nodes to agree on when to stop. Proving part (a), on the other hand, needs more effort. Recall we use the fraction of clear message slots to determine the accuracy of an estimate, and the expected fraction value should be identical for all nodes. However, due to random chances, the actual fraction value observed by each node might deviate from expectation. If we have an upper bound N of n, then by making the first phase to contain $\Theta(\lg N)$ slots, Chernoff bounds [17] will enforce the observed fraction value to be tightly concentrated around its expectation. In our case, N is not available, and we rely on more careful analysis. Specifically, during iterations one to $\lg(n/(a\ln n))$ where a is some sufficiently large constant, in each time slot in phase one, at least two nodes will broadcast (since the estimate is too small), thus no node will obtain private estimate in these iterations. Starting from iteration $\lg(n/(a\ln n))$, the length of phase one is long enough so that concentration inequalities will ensure the observed fraction value is close to its expectation. Building on these observations, we can eventually conclude the following theorem (again, see [20] for full analysis):

▶ **Theorem 9.** *The* Count-SH-noCD-High *approximate neighbor counting algorithm ensures the following properties with high probability in n when executed in a single-hop network with no collision detection: (a) all nodes terminate simultaneously within $O(\lg^2 n)$ slots; and (b) all nodes obtain the same estimate of n, which is in the range $[n, 4n]$.*

5.2 Single-Hop Networks: Collision Detection

Without collision detection, the feedback to a listening node is either silence or a message. Failing to receive a message, therefore, does not hint the nature of the failure: either no node is sending, or multiple nodes are sending. As a result, in the two previous algorithms, when nodes "guess" the count, they have to do it in a *linear* manner: start with a small estimate, and double if the guess is incorrect. With collision detection, by contrast, listening nodes can distinguish whether too few (i.e., zero) or too many (i.e., at least two) nodes are broadcasting. As first pointed out back in the 1980's [22], this extra power enables an exponential improvement over linear searching, since nodes can now perform a *binary search*.

Constant probability of success. Here we leverage the aforementioned binary search strategy to return a constant factor estimate of n in $O(\lg\lg n)$ time, with at least some constant probability. Recall efficient binary search requires a rough upper bound of n as input. To this end, we first introduce an algorithm called EstUpper-SH: it can provide a polynomial upper bound of n within $O(\lg\lg n)$ time. At a high level, EstUpper-SH is doing a linear "guess and verify" search to estimate $\lg n$. (Notice, it is *not* estimating n.) This strategy, to the best of our knowledge, is first discussed by Willard in the seminal paper [22], and has later been used in other works (see, e.g., [5, 3]). Due to space constraint, detailed description and analysis of EstUpper-SH are provided in full version of the paper [20].

Once this estimate is obtained, we switch to the main logic of Count-SH-CD-Const. This algorithm contains multiple iterations, each of which has four time slots. In each iteration i, all nodes have a lower bound a_i and an upper bound b_i, and will test whether the median $m_i = \lfloor (a_i + b_i)/2 \rfloor$ is close to $\lg n$ or not. More specifically, in the first time slot in iteration i, each node will broadcast a beacon message with probability $1/2^{m_i}$, and listen otherwise. Listening nodes will use the channel status they observed to adjust a_i (or b_i), or terminate and output the final estimate. On the other hand, the other three time slots in each iteration allow nodes that have chosen to broadcast in the first time slot to learn the channel status too, with the help of the nodes that have chosen to listen in the first time slot. (See full paper [20] for complete description of Count-SH-CD-Const.)

To prove Count-SH-CD-Const can provide a correct estimate, we demonstrate that during one execution of Count-SH-CD-Const: (a) whenever m_i is too large or too small, all nodes can correctly detect this and adjust a_i or b_i accordingly; and (b) when m_i is a good estimate, all nodes can correctly detect this as well and stop execution. The full analysis can be found in the full version of the paper [20], here we state only the main theorem:

▶ **Theorem 10.** *The* Count-SH-CD-Const *approximate neighbor counting algorithm ensures the following properties when executed in a single-hop network with collision detection: (a) all nodes terminate simultaneously; and (b) with at least constant probability, all nodes obtain the same estimate of n in the range $[n, 4n]$ within $O(\lg\lg n)$ time slots.*

High probability of success. Our last algorithm for the single-hop scenario is Count-SH-CD-High. It significantly differs from the other algorithms studied so far in that it does *not* use a "guess and verify" strategy. Instead, it deploys a *random walk* to derive an estimate. The use of random walks for contention resolution was introduced by Nakano and Olariu [18], in the context of leader election in radio networks. It was later adopted by Brandes et al. [3] for solving approximate counting in beeping networks.

Prior to executing Count-SH-CD-High, nodes will first use $O(\lg\lg n)$ time slots to run EstUpper-SH to obtain a polynomial upper bound of n. Call this upper bound $\hat{N}$. All nodes then perform a random walk, the state space of which consists of potential estimates of n. More specifically, Count-SH-CD-High contains $\Theta(\lg \hat{N})$ iterations, each of which has three time slots. In each iteration, all nodes maintain a current estimate on n which is denoted by $\hat{n}$. (Initially, $\hat{n}$ is set to $\hat{N}$.) In the first slot in a iteration, each node will broadcast a beacon message with probability $1/\hat{n}$, and listen otherwise. If a node hears silence, it will decrease $\hat{n}$ by a factor of four; if a node hears noise, it will increase $\hat{n}$ by a factor of four; and if a node hears a beacon message, it will keep $\hat{n}$ unchanged. Similar to what we have done in Count-SH-CD-Const, in each iteration, the nodes that have chosen to listen in the first time slot will use the latter two slots to help nodes that have chosen to broadcast in the first time slot to learn the channel status of the first time slot. After these $\Theta(\lg \hat{N})$ iterations, all nodes will use $4\tilde{n}$ to be the final estimate of n, where $\tilde{n}$ is the most frequent estimate used by the nodes during the $\Theta(\lg \hat{N})$ iterations.

The high-level intuition of Count-SH-CD-High is: when the estimate is too large or too small, it will quickly shift towards correct estimates; and when the estimate is correct, it will remain unchanged. Therefore, the most frequent estimate will likely to be a correct one. Complete analysis is deferred to full paper [20], here we provide only the main theorem:

▶ **Theorem 11.** *The* Count-SH-CD-High *approximate neighbor counting algorithm ensures the following properties when executed in a single-hop network with collision detection: (a) all nodes terminate simultaneously; and (b) with high probability in n, all nodes obtain the same estimate of n in the range $[n, 64n]$ within $O(\lg n)$ time slots.*

5.3 Multi-Hop with All Nodes Counting: No Collision Detection

All nodes counting in a multi-hop network is challenging as different nodes may have significantly different number of neighbors. In this part, we present two algorithms that attempt to overcome this obstacle, the second of which is particularly interesting.

We begin with the first algorithm – called COUNT-ALL-NOCD– which still relies on the linear "guess and verify" approach. However, it requires the upper bound N_Δ as an input parameter to enforce termination. Nonetheless, for each node, COUNT-ALL-NOCD always returns an accurate estimate, even when knowledge of N_Δ is absent.

In more detail, COUNT-ALL-NOCD contains $\lg N_\Delta$ iterations, and the ith iteration contains $\Theta(i)$ time slots. In each slot in iteration i, each node will choose to be a broadcaster or a listener each with probability $1/2$. If a node chooses to be a listener in a time slot, it will simply listen. Otherwise, if a node chooses to be a broadcaster, it will broadcast a beacon message with probability $1/2^i$, and do nothing otherwise. After an iteration i, for a node u, if for the first time since the beginning of protocol execution, it has heard beacon messages in at least $1/10$ fraction of slots among the listening slots (within this iteration), then u will use 2^{i+3} as its estimate for n_u. Proving the correctness of COUNT-ALL-NOCD borrows heavily from our analysis of COUNT-SH-NOCD-HIGH (see the full version of the paper [20] for more details), here we only state the main theorem:

▶ **Theorem 12.** *For each node u, the* COUNT-ALL-NOCD *approximate neighbor counting algorithm ensures the following with high probability in n_u when executed in a multi-hop network with no collision detection: u will obtain an estimate of n_u in the range $[n_u, 4n_u]$ within $O(\lg^2 n_u)$ time. Moreover, u will terminate after $O(\lg^2 N_\Delta)$ time when N_Δ is known.*

Notice that in COUNT-ALL-NOCD, for a node u, the high correctness guarantee is with respect to n_u. This implies, when n_u is some constant, the probability that the obtained estimate is desirable is also a constant. Sometimes, we may want *identical* and *high* correctness guarantees for *all* estimates, such as high probability in n. Our second algorithm – which is called COUNT-ALL-NOCD-2– achieves this goal, at the cost of accessing N and demanding longer execution time. (COUNT-ALL-NOCD only needs N_Δ to enforce termination, while COUNT-ALL-NOCD-2 needs N to work properly.)

Careful readers might suspect COUNT-ALL-NOCD-2 just extends the length of each iteration of COUNT-ALL-NOCD to $\Theta(\lg N)$. Unfortunately, this simple modification is not sufficient: we still cannot change the fact that when a node u listens, the number of broadcasters among its neighbors is concentrated to $n_u/2$ only with high probability in n_u.

Instead, COUNT-ALL-NOCD-2 takes a different approach, the core of which is a "double counting" trick. Specifically, COUNT-ALL-NOCD-2 contains $L = \Theta(\lg N)$ iterations, each of which has $\Theta(\lg^2 N)$ time slots. At the beginning of each iteration, each node chooses to be a broadcaster or a listener each with probability $1/2$. Then, by applying the "guess and verify" strategy, each listener will spend the $\Theta(\lg^2 N)$ time slots in this iteration to obtain a constant factor estimate on the number of neighboring broadcasters. When all $\Theta(\lg N)$ iterations are done, each node will sum the estimates it has obtained, divide it by $L/4$, and output the result as its estimate for the neighborhood size.

Due to space constraint, detailed description for each iteration is deferred to the full paper [20]. We only note here that the estimates obtained by the listeners are quite accurate:

▶ **Lemma 13.** *Consider an arbitrary iteration during the execution of* COUNT-ALL-NOCD-2*, assume node u is a listener with m neighboring broadcasters. By the end of this iteration, node u will obtain an estimate of m in the range $[m, 4m]$, with high probability in N.*

We can now state and prove the guarantees provided by COUNT-ALL-NOCD-2:

▶ **Theorem 14.** *The* COUNT-ALL-NOCD-2 *approximate neighbor counting algorithm ensures the following properties when executed in a multi-hop network with no collision detection: (a) all nodes terminate after $O(\lg^3 N)$ time slots; and (b) with high probability in N, for each node u, the node will obtain an estimate of n_u in the range $[n_u, 5n_u]$.*

Proof sketch. Consider a node u and one of its neighbor v. Assume COUNT-ALL-NOCD-2 contains $a \lg N$ iterations, where a is a sufficiently large constant. In expectation, in $(a/4) \cdot \lg N$ iterations, u will be listener and v will be broadcaster. Apply a Chernoff bound, we know u will be listener and v will be broadcaster in at least $(1-\delta) \cdot (a/4) \cdot \lg N$ iterations, and at most $(1+\delta) \cdot (a/4) \cdot \lg N$ iteration, w.h.p. in N. Here, $0 < \delta < 1$ is a small constant determined by a. Take a union bound over all $O(N)$ neighbors of u, we know this claim holds true for them as well. Therefore, if u were able to accurately count the number of broadcasting neighbors without any error in each listening iteration, the sum it will obtain would be in the range $[(1-\delta) \cdot (a/4) \cdot \lg N \cdot n_u, (1+\delta) \cdot (a/4) \cdot \lg N \cdot n_u]$, w.h.p. in N.

Now, due to Lemma 13, we know the actual sum of counts u will obtain is in the range $[(1-\delta) \cdot (a/4) \cdot \lg N \cdot n_u, 4 \cdot (1+\delta) \cdot (a/4) \cdot \lg N \cdot n_u]$, w.h.p. in N. As a result, according to our algorithm description, the final estimate u will obtain is in the range $[n_u, 5n_u]$, w.h.p. in N. Take a union bound over all nodes, the theorem is proved. ◀

5.4 Multi-Hop with All Nodes Counting: Collision Detection

In COUNT-ALL-NOCD, we resolve the termination detection problem by accessing N_Δ. This allows nodes to run long enough so that they could be sure that everyone has a chance to learn what it needed to learn. With the addition of collision detection, however, the assumption that nodes know N_Δ can be removed for many network topologies. In particular, we can leverage the idea that neighbors of u that *have not* obtained estimates yet can use noise to *reliably* inform u that they wish u to continue. We call this algorithm COUNT-ALL-CD.

COUNT-ALL-CD contains multiple iterations, each of which has two parts. The first part of any iteration i is identical to iteration i of COUNT-ALL-NOCD. The second part, on the other hand, helps nodes to determine when to stop. In particular, the second part of iteration i contains a single slot. For a node u, if it has not obtained an estimate of n_u by the end of the first part of iteration i yet, then in the second part, it will broadcast a continue message. On the other hand, if u has already obtained an estimate of n_u by the end of the first part of iteration i, it will simply listen in part two. Moreover, u will continue into the next iteration iff it hears continue or noise during part two. The guarantees provided by COUNT-ALL-CD are stated below, and the proof of it is provided in the full paper [20].

▶ **Theorem 15.** *The* COUNT-ALL-NOCD *approximate neighbor counting algorithm ensures the following properties for each node u when executed in a multi-hop network with collision detection: (a) u will obtain an estimate of n_u in the range $[n_u, 4n_u]$ within $O(\lg^2 n_u)$ time slots, with high probability in n_u; and (b) if $\sum_{v \in \Gamma_u \cup \{u\}} (1/n_v) < 1$, then u will terminate within $(\max_{v \in \Gamma_u \cup \{u\}} \{\lg n_v\})^2$ time slots, with probability at least $1 - \sum_{v \in \Gamma_u \cup \{u\}} (1/n_v)$.*

A key point about the above termination bound is that it requires $\sum_{v \in \Gamma_u \cup \{u\}} (1/n_v) < 1$. (In fact, this constraint can be relaxed to $\sum_{v \in \Gamma_u \cup \{u\}} (1/n_v^\gamma) < 1$, for an arbitrarily chosen constant $\gamma \geq 1$.) If this is not the case (e.g., in a dense star network), the termination detection mechanism might not work properly. In that situation, the default dependence on N_Δ from the no collision detection case can be applied as a back-up.

5.5 Multi-Hop with Designated Node Counting

Compared to the all nodes counting variant, multi-hop neighbor counting with only the designated node is easier: the strategies we previously used for single-hop counting are still applicable, and the introduction of the designated node can actually make coordination easier. (In particular, this node can greatly simplify termination detection). Due to space constraint, we defer the upper bounds for this variant to the full version of the paper [20].

We note that one interesting algorithm in this variant is Count-Desig-noCD-Const, which achieves constant success probability without collision detection. Count-Desig-noCD-Const differs from its single-hop counterpart (i.e., Count-SH-noCD-Const) in that it uses fraction of clear message slots to determine the accuracy of nodes' estimate. The primary reason we develop this algorithm is that the success probability of Count-SH-noCD-Const is *fixed*. In contrast, in Count-Desig-noCD-Const, by tweaking the running time (up to some constant factor), the success probability – despite being a constant – can be adjusted accordingly.

6 Discussion

We see at least two problems that worth further exploration. First, how do termination requirements affect the complexity of the problem? This is particularly interesting in the multi-hop all nodes counting scenario, when knowledge of N_Δ or N is not available. Count-All-noCD shows lower bound can be achieved at the cost of no termination, but how much time must we spend if termination needs to be enforced, or is simply impossible without knowing N_Δ or N? Another open problem concerns the gap between the lower and upper bounds, in the multi-hop all nodes counting scenario with collision detection. On the one hand, the lower bound might be loose as it is a simple carry over, ignoring the possibility that all nodes counting could be fundamentally harder than designated node counting. Yet on the other hand, we have not found a way to leverage collision detection to reduce algorithm runtime (e.g., it seems hard to run multiple instances of binary search in parallel). Currently, our best guess is that *both* the lower bound and the upper bound are not tight.

References

1 Reuven Bar-Yehuda, Oded Goldreich, and Alon Itai. On the Time-complexity of Broadcast in Radio Networks: An Exponential Gap Between Determinism and Randomization. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 98–108. ACM, 1987.

2 Jacir Luiz Bordim, JiangTao Cui, Tatsuya Hayashi, Koji Nakano, and Stephan Olariu. Energy-Efficient Initialization Protocols for Ad-hoc Radio Networks. In *International Symposium on Algorithms and Computation*, ISAAC '99, pages 215–224. Springer, 1999.

3 Philipp Brandes, Marcin Kardas, Marek Klonowski, Dominik Pajak, and Roger Wattenhofer. Fast Size Approximation of a Radio Network in Beeping Model. *Theoretical Computer Science*, In Press, 2017. doi:10.1016/j.tcs.2017.05.022.

4 Ioannis Caragiannis, Clemente Galdi, and Christos Kaklamanis. Basic Computations in Wireless Networks. In *International Symposium on Algorithms and Computation*, ISAAC '05, pages 533–542. Springer, 2005.

5 Binbin Chen, Ziling Zhou, and Haifeng Yu. Understanding RFID Counting Protocols. In *Proceedings of the 19th Annual International Conference on Mobile Computing and Networking*, MobiCom '13, pages 291–302. ACM, 2013.

6 Israel Cidon and Moshe Sidi. Conflict Multiplicity Estimation and Batch Resolution Algorithms. *IEEE Transactions on Information Theory*, 34(1):101–110, 1988.

7 Alejandro Cornejo and Fabian Kuhn. Deploying Wireless Networks with Beeps. In *International Symposium on Distributed Computing*, DISC '10, pages 148–162. Springer, 2010.

8 Mohsen Ghaffari, Nancy Lynch, and Srikanth Sastry. Leader Election Using Loneliness Detection. *Distributed Computing*, 25(6):427–450, 2012.

9 Seth Gilbert, Valerie King, Seth Pettie, Ely Porat, Jared Saia, and Maxwell Young. (Near) Optimal Resource-competitive Broadcast with Jamming. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, pages 257–266. ACM, 2014.

10 Seth Gilbert, Fabian Kuhn, and Chaodong Zheng. Communication Primitives in Cognitive Radio Networks. In *Proceedings of the 2017 ACM Symposium on Principles of Distributed Computing*, PODC '17, pages 23–32. ACM, 2017.

11 Albert G. Greenberg, Philippe Flajolet, and Richard E. Ladner. Estimating the Multiplicities of Conflicts to Speed Their Resolution in Multiple Access Channels. *Journal of the ACM*, 34(2):289–325, 1987.

12 Tomasz Jurdziński, Mirosław Kutyłowski, and Jan Zatopiański. Energy-Efficient Size Approximation of Radio Networks with No Collision Detection. In *International Computing and Combinatorics Conference*, COCOON '02, pages 279–289. Springer, 2002.

13 Tomasz Jurdzinski and Grzegorz Stachowiak. Probabilistic Algorithms for the Wake-Up Problem in Single-Hop Radio Networks. *Theory of Computing Systems*, 38(3):347–367, 2005.

14 Jedrzej Kabarowski, Mirosław Kutyłowski, and Wojciech Rutkowski. Adversary Immune Size Approximation of Single-Hop Radio Networks. In *International Conference on Theory and Applications of Models of Computation*, pages 148–158. Springer, 2006.

15 Marek Klonowski and Kamil Wolny. Immune Size Approximation Algorithms in Ad Hoc Radio Network. In *European Conference on Wireless Sensor Networks*, pages 33–48. Springer, 2012.

16 Michael Luby. A Simple Parallel Algorithm for the Maximal Independent Set Problem. In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing*, STOC '85, pages 1–10. ACM, 1985.

17 Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

18 Koji Nakan and Stephan Olari. Uniform Leader Election Protocols for Radio Networks. *IEEE Transactions on Parallel and Distributed Systems*, 13(5):516–526, 2002.

19 Calvin Newport. Radio Network Lower Bounds Made Easy. In *Proceedings of the 28th International Symposium on Distributed Computing*, DISC '14, pages 258–272. Springer, 2014.

20 Calvin Newport and Chaodong Zheng. Approximate Neighbor Counting in Radio Networks, 2018. `arXiv:1811.03278`.

21 Muhammad Shahzad and Alex X. Liu. Every Bit Counts: Fast and Scalable RFID Estimation. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, MobiCom '12, pages 365–376. ACM, 2012.

22 Dan E. Willard. Log-logarithmic Selection Resolution Protocols in a Multiple Access Channel. *SIAM Journal on Computing*, 15(2):468–477, 1986.

23 Yuanqing Zheng and Mo Li. ZOE: Fast cardinality estimation for large-scale RFID systems. In *Proceedings of the 32nd IEEE International Conference on Computer Communications*, INFOCOM '13, pages 908–916. IEEE, 2013.

24 Yuanqing Zheng, Mo Li, and Chen Qian. PET: Probabilistic Estimating Tree for Large-Scale RFID Estimation. In *Proceedings of the 31st International Conference on Distributed Computing Systems*, ICDCS '11, pages 37–46. IEEE, 2011.

On Simple Back-Off in Unreliable Radio Networks*

Seth Gilbert
National University of Singapore, Singapore
seth.gilbert@comp.nus.edu.sg

Nancy Lynch
MIT, Cambridge, MA, USA
lynch@csail.mit.edu

Calvin Newport
Georgetown University, Washington, DC, USA
cnewport@cs.georgetown.edu

Dominik Pajak
MIT, Cambridge, MA, USA
pajak@csail.mit.edu

Abstract

In this paper, we study local and global broadcast in the dual graph model, which describes communication in a radio network with both reliable and unreliable links. Existing work proved that efficient solutions to these problems are impossible in the dual graph model under standard assumptions. In real networks, however, simple back-off strategies tend to perform well for solving these basic communication tasks. We address this apparent paradox by introducing a new set of constraints to the dual graph model that better generalize the slow/fast fading behavior common in real networks. We prove that in the context of these new constraints, simple back-off strategies now provide efficient solutions to local and global broadcast in the dual graph model. We also precisely characterize how this efficiency degrades as the new constraints are reduced down to non-existent, and prove new lower bounds that establish this degradation as near optimal for a large class of natural algorithms. We conclude with an analysis of a more general model where we propose an enhanced back-off algorithm. These results provide theoretical foundations for the practical observation that simple back-off algorithms tend to work well even amid the complicated link dynamics of real radio networks.

2012 ACM Subject Classification Theory of computation → Distributed algorithms, Networks → Ad hoc networks

Keywords and phrases radio networks, broadcast, unreliable links, distributed algorithm, robustness

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.27

Related Version The full version is available at https://arxiv.org/abs/1803.02216.

* Definitions and preliminary results concerning the local broadcast problem appeared in the brief announcement [10], published in the Proceedings of 32nd International Symposium on DIStributed Computing (DISC) 2018.

© Seth Gilbert, Nancy Lynch, Calvin Newport, and Dominik Pajak;
licensed under Creative Commons License CC-BY
22nd International Conference on Principles of Distributed Systems (OPODIS 2018).
Editors: Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira; Article No. 27; pp. 27:1–27:17

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

1 Introduction

In this paper, we study upper and lower bounds for efficient broadcast in the dual graph radio network model [4, 11, 12, 3, 6, 5, 8, 7, 14, 9], a dynamic network model that describes wireless communication over both reliable and unreliable links. As argued in previous studies of this setting, including unpredictable link behavior in theoretical wireless network models is important because in real world deployments radio links are often quite dynamic.

The Back-Off Paradox. Existing papers [12, 8, 14] proved that it is impossible to solve standard broadcast problems efficiently in the dual graph model without the addition of strong extra assumptions (see related work). In real radio networks, however, which suffer from the type of link dynamics abstracted by the dual graph model, simple back-off strategies tend to perform quite well. These dueling realities seem to imply a dispiriting gap between theory and practice: basic communication tasks that are easily solved in real networks are impossible when studied in abstract models of these networks.

What explains this paradox? This paper tackles this fundamental question.

As detailed below, we focus our attention on the *adversary* entity that decides which unreliable links to include in the network topology in each round of an execution in the dual graph model. We introduce a new type of adversary with constraints that better generalize the dynamic behavior of real radio links. We then reexamine simple back-off strategies originally introduced in the standard radio network model [2] (which has only reliable links), and prove that for reasonable parameters, these simple strategies *now do* guarantee efficient communication in the dual graph model combined with our new, more realistic adversary.

We also detail how this performance degrades toward the existing dual graph lower bounds as the new constraints are reduced toward non-existent, and prove lower bounds that establish these bounds to be near tight for a large and natural class of back-off strategies. Finally, we perform investigations of even more general (and therefore more difficult) variations of this new style of adversary that continue to underscore the versatility of simple back-off strategies.

We argue that these results help resolve the back-off paradox described above. When unpredictable link behavior is modeled properly, predictable algorithms prove to work surprisingly well.

The Dual Graph Model. The dual graph model describes a radio network topology with two graphs, $G = (V, E)$ and $G' = (V, E')$, where $E \subseteq E'$, V corresponds to the wireless devices, E corresponds to reliable (high quality) links, and $E' \setminus E$ corresponds to unreliable (quality varies over time) links. In each round, all edges from E are included in the network topology. Also included is an additional subset of edges from $E' \setminus E$, chosen by an *adversary*. This subset can change from round to round. Once the topology is set for the round, the model implements the standard communication rules from the classical radio network model: a node u receives a message broadcast by its neighbor v in the topology if and only if u decides to receive and v is its only neighbor broadcasting in the round.

We emphasize that the abstract models used in the sizable literature studying distributed algorithms in wireless settings do not claim to provide high fidelity representations of real world radio signal communication. They instead each capture core dynamics of this setting, enabling the investigation of fundamental algorithmic questions. The well-studied radio network model, for example, provides a simple but instructive abstraction of message loss due to collision. The dual graph model generalizes this abstraction to also include network topology dynamics. Studying the gaps between these two models provides insight into the hardness induced by the types of link quality changes common in real wireless networks.

The Fading Adversary. Existing studies of the dual graph model focused mainly on the information about the algorithm known to the model adversary when it makes its edge choices. In this paper, we place additional constraints on how these choices are generated.

In more detail, in each round, the adversary independently draws the set of edges from $E' \setminus E$ to add to the topology from some probability distribution defined over this set. We do not constrain the properties of the distributions selected by the adversary. Indeed, it is perfectly valid for the adversary in a given round to use a point distribution that puts the full probability mass on a single subset, giving it full control over its selection for the round. We also assume the algorithm executing in the model has no advance knowledge of the distributions used by the adversary.

We do, however, constrain how often the adversary can change the distribution from which it selects these edge subsets. In more detail, we parameterize the model with a *stability factor*, $\tau \geq 1$, and restrict the adversary to changing the distribution it uses at most once every τ rounds. For $\tau = 1$, the adversary can change the distribution in every round, and is therefore effectively unconstrained and behaves the same as in the existing dual graph studies. On the other extreme, for $\tau = \infty$, the adversary is now quite constrained in that it must draw edges independently from the same distribution for the entire execution. As detailed below, we find $\tau \approx \log \Delta$, for local neighborhood size Δ, to be a key threshold after which efficient communication becomes tractable.

Notice, these constraints do not prevent the adversary from inducing large amounts of changes to the network topology from round to round. For non-trivial τ values, however, they do require changes that are nearby in time to share some underlying stochastic structure. This property is inspired by the general way wireless network engineers think about unreliability in radio links. In their analytical models of link behavior (used, for example, to analyze modulation or rate selection schemes, or to model signal propagation in simulation), engineers often assume that in the short term, changes to link quality come from sources like noise and multi-path effects, which can be approximated by independent draws from an underlying distribution (Gaussian distributions are common choices for this purpose). Long term changes, by contrast, can come from modifications to the network environment itself, such as devices moving, which do not necessarily have an obvious stochastic structure, but unfold at a slower rate than short term fluctuations.

In our model, the distribution used in a given round captures short term changes, while the adversary's arbitrary (but rate-limited) changes to these distributions over time capture long term changes. Because these general types of changes are sometimes labeled *short/fast fading* in the systems literature (e.g., [16]), we call our new adversary a *fading adversary.*

Our Results and Related Work. In this paper, we study both *local* and *global* broadcast. The local version of this problems assumes some subset of devices in a dual graph network are provided broadcast messages. The problem is solved once each receiver that neighbors a broadcaster in E receives at least one message. The global version assumes a single broadcaster starts with a message that it must disseminate to the entire network. Below we summarize the relevant related work on these problems, and the new bounds proved in this paper. We conclude with a discussion of the key ideas behind these new results.

Related Work. In the standard radio network model, which is equivalent to the dual graph model with $E = E'$, Bar-Yehuda et al. [2] demonstrate that a simple randomized back-off strategy called *Decay* solves local broadcast in $O(\log^2 n)$ rounds and global broadcast in $O(D \log n + \log^2 n)$ rounds, where $n = |V|$ is the network size and D is the diameter of G. Both results hold with high probability in n, and were subsequently proved to be optimal

Table 1 A summary of the upper and lower bounds proved in this paper, along with pointers to the corresponding theorems. In the following, n is the network size, $\Delta \leq n$ is an upper bound on local neighborhood size, D is the (reliable link) network diameter, and τ is the stability factor constraining the adversary.

Problem	Time	Prob.	Remarks	Ref.
Local broadcast	$O\left(\frac{\Delta^{1/\bar{\tau}} \cdot \bar{\tau}^2}{\log \Delta} \cdot \log(1/\epsilon)\right)$	$1-\epsilon$	$\bar{\tau} = \min\{\tau, \log \Delta\}$	Thm 6
	$\Omega\left(\frac{\Delta^{1/\tau}\tau}{\log \Delta}\right)$	$\frac{1}{2}$	$\tau \in O(\log \Delta)$	Thm 7
	$\Omega\left(\frac{\Delta^{1/\tau}\tau^2}{\log \Delta}\right)$	$\frac{1}{2}$	$\tau \in O(\log \Delta / \log\log \Delta)$	Thm 8
Global broadcast	$O\left((D + \log(n/\epsilon)) \cdot \frac{\Delta^{1/\bar{\tau}}\bar{\tau}^2}{\log \Delta}\right)$	$1-\epsilon$	$\bar{\tau} = \min\{\tau, \log \Delta\}$	Thm 9
	$\Omega\left(D \cdot \frac{\Delta^{1/\tau}\tau}{\log \Delta}\right)$	$\frac{1}{2}$	$\tau \in O(\log \Delta)$	Thm 10
	$\Omega\left(D \cdot \frac{\Delta^{1/\tau}\tau^2}{\log \Delta}\right)$	$\frac{1}{2}$	$\tau \in O(\log \Delta / \log\log \Delta)$	Thm 10

or near optimal[1] [1, 13, 15]. In [11, 12], it is proved that global broadcast (with constant diameter), and local broadcast require $\Omega(n)$ rounds to solve with reasonable probability in the dual graph model with an offline adaptive adversary controlling the unreliable edge selection, while [8] proves that $\Omega(n/\log n)$ rounds are necessary for both problems with an online adaptive adversary. As also proved in [8]: even with the weaker oblivious adversary, local broadcast requires $\Omega(\sqrt{n}/\log n)$ rounds, whereas global broadcast *can* be solved in an efficient $O(D \log(n/D) + \log^2 n)$ rounds, but only if the broadcast message is sufficiently large to contain enough shared random bits for all nodes to use throughout the execution. In [14], an efficient algorithm for local broadcast with an oblivious adversary is provided given the assumption of geographic constraints on the dual graphs, enabling complicated clustering strategies that allow nearby devices to coordinate randomness.

New Results. In this paper, we turn our attention to local and global broadcast in the dual graph model with a fading adversary constrained by some stability factor τ (unknown to the algorithm). We start by considering upper bounds for a simple back-off style strategy inspired by the *decay* routine from [2]. This routine has broadcasters simply cycle through a fixed set of broadcast probabilities in a synchronized manner (all broadcasters use the same probability in the same round). We prove that this strategy solves local broadcast with probability at least $1-\epsilon$, in $O\left(\frac{\Delta^{1/\bar{\tau}} \cdot \bar{\tau}^2}{\log \Delta} \cdot \log(1/\epsilon)\right)$ rounds, where Δ is an upper bound on local neighborhood size, and $\bar{\tau} = \min\{\tau, \log \Delta\}$.

Notice, for $\tau \geq \log \Delta$ this bound simplifies to $O(\log \Delta \log(1/\epsilon))$, matching the optimal results from the standard radio network model.[2] This performance, however, degrades toward the polynomial lower bounds from the existing dual graph literature as τ reduces from $\log \Delta$ toward a minimum value of 1. We show this degradation to be near optimal by proving that *any* local broadcast algorithm that uses a fixed sequence of broadcast probabilities requires $\Omega(\Delta^{1/\tau}\tau/\log \Delta)$ rounds to solve the problem with probability 1/2 for a given τ. For $\tau \in O(\log \Delta / \log\log \Delta)$, we refine this bound further to $\Omega(\Delta^{1/\tau}\tau^2/\log \Delta)$, matching our upper bound within constant factors.

---

[1] The broadcast algorithm from [2] requires $O(D \log n + \log^2 n)$ rounds, whereas the corresponding lower bound is $\Omega(D \log(n/D) + \log^2 n)$. This gap was subsequently closed by a tighter analysis of a natural variation of the simple *Decay* strategy used in [2]

[2] To make it match exactly, set $\Delta = n$ and $\epsilon = 1/n$, as is often assumed in this prior work.

We next turn our attention to global broadcast. We consider a straightforward global broadcast algorithm that uses our local broadcast strategy as a subroutine. We prove that this algorithm solves global broadcast with probability at least $1-\epsilon$, in $O(D+\log(n/\epsilon))\cdot\Delta^{1/\bar{\tau}}\bar{\tau}^2/\log\Delta)$ rounds, where D is the diameter of G, and $\bar{\tau}=\min\{\tau,\log\Delta\}$. Notice, for $\tau\geq\log\Delta$ this bound reduces to $O(D\log\Delta+\log\Delta\log(1/\epsilon))$, matching the near optimal result from the standard radio network model. As with local broadcast, we also prove the degradation of this performance as τ shrinks to be near optimal. (See Table 1 for a summary of these results and pointers to where they are proved in this paper.)

Finally we consider the generalized model when we allow correlation between the distributions selected by the adversary within a given stable period of τ rounds. It turns out that in the case of arbitrary correlations any simple algorithm needs time $\Omega(\sqrt{\Delta}/l)$ if it uses only cycles of length l. In particular any our previous algorithms would require time $\Omega(\sqrt{\Delta}/\log\Delta)$ in the model with arbitrary correlations. The adversary construction in this lower bound requires large changes in the degree of a node in successive steps. Such changes are unlikely in real networks thus we propose a restricted version of the adversary. We assume that the expected change in the degree of any node can be at most $\Delta^{1/(\bar{\tau}(1-o(1))}$. With such restriction it is again possible to propose a simple, but slightly enhanced, back-off strategy (with a short cycle of probabilities) that works efficiently in time $O\left(\Delta^{1/\bar{\tau}}\cdot\bar{\tau}\cdot\log(1/\epsilon)\right)$.

Technique Discussion. Simple back-off strategies can be understood as experimenting with different *guesses* at the amount of contention afflicting a given receiver. If the network topology is static, this contention is fixed, therefore so is the *right* guess. A simple strategy cycling through a reasonable set of guesses will soon arrive at this right guess – giving the message a good chance of propagating.

The existing lower bounds in the dual graph setting deploy an adversary that changes the topology in each round to specifically thwart that round's guess. In this way, the algorithm never has the right guess for the current round so its probability of progress is diminished. The fading adversary, by contrast, is prevented from adopting this degenerate behavior because it is required to stick with the same distribution for τ consecutive rounds. An important analysis at the core of our upper bounds reveals that any fixed distribution will be associated with a right guess defined with respect to the details of that distribution. If τ is sufficiently large, our algorithms are able to experiment with enough guesses to hit on this right guess before the adversary is able to change the distribution.

More generally speaking, the difficulty of broadcast in the previous dual graph studies was *not* due to the ability of the topology to change dramatically from round to round (which can happen in practice), but instead due to the model's ability to precisely tune these changes to thwart the algorithm (a behavior that is hard to motivate). The dual graph model with the fading adversary preserves the former (realistic) behavior while minimizing the latter (unrealistic) behavior.

2 Model and Problem

We study the dual graph model of unreliable radio networks. This model describes the network topology with two graphs $G=(V,E)$ and $G'=(V,E')$, where $E\subseteq E'$. The $n=|V|$ vertices in V correspond to the wireless devices in the network, which we call *nodes* in the following. The edge in E describe reliable links (which maintain a consistently high quality), while the edges in $E'\setminus E$ describe unreliable links (which have quality that can vary over time). For a given dual graph, we use Δ to describe the maximum degree in G', and D to describe the diameter of G.

Time proceeds in synchronous rounds that we label $1, 2, 3...$ For each round $r \geq 1$, the network topology is described by $G_r = (V, E_r)$, where E_r contains all edges in E plus a subset of the edges in $E' \setminus E$. The subset of edges from $E' \setminus E$ are selected by an *adversary*. The graph G_r can be interpreted as describing the high quality links during round r. That is, if $\{u, v\} \in E_r$, this mean the link between u and v is strong enough that u could deliver a message to v, or garble another message being sent to v at the same time.

With the topology G_r established for the round, behavior proceeds as in the standard radio network model. That is, each node $u \in V$ can decide to transmit or receive. If u transmits, it learns nothing about other messages transmitted in the round (i.e., the radios are half-duplex). If u receives and exactly one neighbor v of u in E_r transmits, then u receives v's message. If u receives and two or more neighbors in E_r transmit, u receives nothing as the messages are lost due to collision. If u receives and no neighbor transmits, u also receives nothing. We assume u does not have collision detection, meaning it cannot distinguish between these last two cases.

The Fading Adversary. A key assumption in studying the dual graph model are the constraints placed on the adversary that selects the unreliable edges to include in the network topology in each round. In this paper, we study a new set of constraints inspired by real network behavior. In more detail, we parameterize the adversary with a *stability factor* that we represent with an integer $\tau \geq 1$. In each round, the adversary must draw the subset of edges (if any) from $E' \setminus E$ to include in the topology from a distribution defined over these edges. The adversary selects which distributions it uses. Indeed, we assume it is *adaptive* in the sense that it can wait until the beginning of a given round before deciding the distribution it will use in that round, basing its decision on the history of the nodes' transmit/receive behavior up to this point, including the previous messages they send, but not including knowledge of the nodes' private random bits.

The adversary is constrained, however, in that it can change this distribution at most once every τ rounds. On one extreme, if $\tau = 1$, it can change the distribution in every round and is effectively unconstrained in its choices. On the other other extreme, if $\tau = \infty$, it must stick with the same distribution for every round. For most of this paper, we assume the draws from these distributions are independent in each round. Toward the end, however, we briefly discuss what happens when we generalize the model to allow more correlations.

As detailed in the introduction, because these constraints roughly approximate the fast/slow fading behavior common in the study of real wireless networks, we call a dual graph adversary constrained in this manner a *fading adversary*.

Problem. In this paper, we study both the *local* and *global* broadcast problems. The local broadcast problem assumes a set $B \subseteq V$ of nodes are provided with a message to broadcast. Each node can receive a unique message. Let $R \subseteq V$ be the set of nodes in V that neighbor at least one node in B in E. The problem is solved once every node in R has received at least one message from a node in B. We assume all nodes in B start the execution during round 1, but do not require that B and R are disjoint (i.e., broadcasters can also be receivers). The global broadcast problem, by contrast, assumes a single source node in V is provided a broadcast message during round 1. The problem is solved once all nodes have received this message. Notice, local broadcast solutions are often used as subroutines to help solve global broadcast.

Uniform Algorithms. The broadcast upper and lower bounds we study in this paper focus on *uniform algorithms*, which require nodes to make their probabilistic transmission decisions according to a predetermined sequence of broadcast probabilities that we express as a repeating cycle, $(p_1, p_2, ..., p_k)$ of k probabilities in synchrony. In studying global broadcast, we assume that on first receiving a message, a node can wait to start making probabilistic transmission decisions until the cycle resets. We assume these probabilities can depend on n, Δ and τ (or worst-case bounds on these values).

In uniform algorithms in the model with fading adversary an important parameter of any node v is its *effective degree* in step t denoted by $d_t(v)$ and defined as the number of nodes w such that $(v, w) \in E_t$ and w has a message to transmit (i.e., participates in step t).

As mentioned in the introduction, uniform algorithms, such as the *decay* strategy from [2], solve local and global broadcast with optimal efficiency in the standard radio network model. A major focus of this paper is to prove that they work well in the dual graph model as well, if we assume a fading adversary with a reasonable stability factor.

The fact that our lower bounds assume the algorithms are uniform technically weaken the results, as there might be more efficient non-uniform strategies. In the standard radio network model, however, this does not prove to be the case: uniform algorithms for local and global broadcast match lower bounds that hold for all algorithms (c.f., discussion in [15]).

3 Local broadcast

We begin by studying upper and lower bounds for the local broadcast problem. Our upper bound performs efficiently once the stability factor τ reaches a threshold of $\log \Delta$. As τ decreases toward a minimum value of 1, this efficiency degrades rapidly. Our lower bounds capture that this degradation for small τ is unavoidable for uniform algorithms. In the following we use the notation $\bar{\tau} = \min\{\tau, \lceil \log \Delta \rceil\}$. By $\log n$ we will always denote logarithm at base 2 and by $\ln n$ the natural logarithm.

3.1 Upper Bound

All uniform local broadcast algorithms behave in the same manner: the nodes in B repeatedly broadcast according to fixed cycle of k broadcast probabilities. We formalize this strategy with algorithm `RLB` (Robust Local Broadcast) described below. We break out `Uniform` into its own procedure as we later reuse it in our improved algorithm:

```
1 Procedure: Uniform(k, p_1, p_2, ..., p_k)
2 for i = 1, 2, ..., k do
3     if has message then
4         with probability p_i Transmit otherwise
          Listen
5     else Listen
```

```
1 Algorithm: RLB(r, τ̄)
2 for i ← 1 to τ̄ do
      p_i ← Δ^(−i/τ̄)
3 repeat r times
4     Uniform
      (τ̄, p_1, p_2, ..., p_τ̄)
```

Before we prove the complexity of `RLB` we will show two useful properties of any uniform algorithm. Let $R_t^{(v)}$ denote the event that v receives a message from its neighbor in step t.

▶ **Lemma 1.** *For any uniform algorithm and any node v and step t if $d_t(v) > 0$ and the algorithm uses in step t probability $p \le 1/2$, then* $\mathbf{Pr}\left[R_t^{(v)}\right] \ge \frac{p \cdot d_t(v)}{(2e)^{p \cdot d_t(v)}}$.

Proof. For this to happen exactly one among $d_t(v)$ neighbors of v has to transmit and v must not transmit. Node v does not transmit with probability $1 - p$ if it has the message

and clearly with probability 1 if it has the message. Denote by $\alpha = p \cdot d_t(v)$. We have

$$\mathbf{Pr}\left[R_t^{(v)}\right] \geq pd_t(v) \cdot (1-p)^{d_t(v)} = \alpha \cdot \left(1 - \frac{\alpha}{d_t(v)}\right)^{d_t(v)}$$
$$= \alpha\left(\left(1 - \frac{\alpha}{d_i(v)}\right)^{d_t(v)/\alpha - 1} \cdot (1-p)\right)^{\alpha} \geq \alpha(e^{-1}(1-p))^{\alpha} \geq \frac{\alpha}{(2e)^{\alpha}}.$$

◀

▶ **Lemma 2.** *For any uniform algorithm, node v and step t if $d_t(v) > 0$:*

$$\mathbf{Pr}\left[R_t^{(v)} \mid d_t(v) \in [d_1, d_2]\right] \geq \min\left\{\mathbf{Pr}\left[R_t^{(v)} \mid d_t(v) = d_1\right], \mathbf{Pr}\left[R_t^{(v)} \mid d_t(v) = d_2\right]\right\}.$$

Proof. If the algorithm uses probability p in step t then $\mathbf{Pr}\left[R_t^{(v)}\right] = pd_t(v)(1-p)^{d_t(v)}$. Seeing this expression as a function of $d_t(v)$ we can compute the derivative and obtain that this function has a single maximum in $d_t(v) = 1/(\ln(1/(1-p)))$. Hence if we restrict $d_t(v)$ to be within a certain interval, then value of the function is lower bounded by the minimum at the endpoints of the interval. ◀

Our upper bound analysis leverages the following useful lemma which can be shown by induction on n (the left side is also known as the Weierstrass Product Inequality):

▶ **Lemma 3.** *For any $x_1, x_2, \ldots, x_n$ such that $0 \leq x_i \leq 1$:*

$$1 - \sum_{i=1}^{n} x_i \leq \prod_{i=1}^{n}(1 - x_i) \leq 1 - \sum_{i=1}^{n} x_i + \sum_{1 \leq i < j \leq n} x_i x_j.$$

To begin our analysis, we focus on the behavior of our algorithm with respect to a single receiver when we use the transmit probability sequence $p_1, p_2, ..., p_{\bar{\tau}}$, where $\bar{\tau} = \min\{\tau, \lceil \log \Delta \rceil\}$, and $p_i = \Delta^{-i/\bar{\tau}}$.

▶ **Lemma 4.** *Fix any receiver $u \in R$ and error bound $\epsilon > 0$. It follows: RLB$(2\lceil \ln(1/\epsilon) \rceil \cdot \lceil 4e \cdot \Delta^{1/\bar{\tau}} \rceil, \bar{\tau})$ delivers a message to u with probability at least $1 - \epsilon$ in time $O(\Delta^{1/\bar{\tau}} \bar{\tau} \log(1/\epsilon))$.*

Proof. It is sufficient to prove the claim for $\tau \leq \log \Delta$. For $\tau > \log \Delta$ we use the algorithm for $\tau = \log \Delta$. Note that any algorithm that is correct for some τ must also work for any larger τ because the adversary may not choose to change the distribution as frequently as it is permitted to. In the case where $\tau \leq \log \Delta$ we get that $\Delta^{1/\tau} \geq 2$.

We want to show that if the nodes from $N_u \cap B$ execute procedure `Uniform`$(\tau, p_1, \ldots, p_\tau)$ twice, then u receives some message with probability at least $\log \Delta/(2e\Delta^{1/\tau}\tau)$. Every time we execute `Uniform` twice, we have a total of 2τ consecutive time slots out of which, by the definition of our model, at least τ consecutive slots have the same distribution of the additional edges and moreover stations try all the probabilities $p_1, p_2, \ldots, p_\tau$ (not necessarily in this order). Let T denote the set of these τ time slots and for $i = 1, 2, \ldots, \tau$ let $t_i \in T$ be the step in which probability p_i is used. We also denote the distribution used in steps from set T by $\mathcal{E}^{(\mathcal{T})}$. Hence we can denote the edges between u and its neighbors that have some message by $E_{part} = \{(u, b) : b \in B\} \cap E'$. We know that the edge sets are chosen independently from the same distribution: $E_t \sim \mathcal{E}^{(\mathcal{T})}$ for $t \in T$. Let us denote by $X_t = |E_t \cap E_{part}|$ the random variable being the number of neighbors that are connected to u in step t and belong to B. For each i from 1 to τ we define $q_i = \mathbf{Pr}\left[\Delta^{(i-1)/\tau} < X_t \leq \Delta^{i/\tau}\right]$, for any $t \in T$. Observe that probabilities q_i do not depend on t during the considered τ

rounds. Moreover since $u \in R$ then u is connected via a reliable edge to at least one node in B, thus $E \cap E_{part} \neq \emptyset$, hence $\mathbf{Pr}[\, X_t = 0\,] = 0$ thus:

$$\sum_{i=1}^{\tau} q_i = 1, \tag{1}$$

Let S_i denote the indicator random variable being 1 if in t_i-th round if exactly one neighbor of u transmits and u is not transmitting in round t and 0 otherwise. Clearly if $S_i = 1$ in some round t, then u receives some message in round t. Then we would like to show for each $i = 1, 2, \dots, \tau$ that:

$$\mathbf{Pr}[\, S_i = 1\,] \geq \frac{q_i}{2e\Delta^{1/\tau}}. \tag{2}$$

In t_i-th slot the transmission probability is $p_i = \Delta^{-i/\tau}$ and the transmission choices done by the stations are independent from the choice of edges E_{t_i} active in round t_i. Note that u might also belong R and try to transmit. But since $p_i \leq 1/2$ then u is not transmitting with probability at least $1/2$. If Q_i denotes the event that $\Delta^{(i-1)/\tau} < X_{t_i} \leq \Delta^{i/\tau}$ then:

$$\begin{aligned}
\mathbf{Pr}[\, S_i = 1\,] &\geq \mathbf{Pr}[\, S_i = 1 | Q_i\,] \cdot \mathbf{Pr}[\, Q_i\,] \geq p_i(\Delta^{(i-1)/\tau} + 1) \cdot (1 - p_i)^{\Delta^{(i-1)/\tau}} \cdot \frac{1}{2} \cdot q_i \\
&\geq p_i \Delta^{(i-1)/\tau} \cdot (1 - p_i)^{\Delta^{i/\tau} - 1} \cdot \frac{1}{2} \cdot q_i \\
&\geq \Delta^{-1/\tau} \cdot \left(1 - \frac{1}{\Delta^{i/\tau}}\right)^{\Delta^{i/\tau} - 1} \cdot \frac{q_i}{2} \geq \frac{q_i}{2e\Delta^{1/\tau}},
\end{aligned}$$

because inequality $(1 - 1/x)^{x-1} \geq e^{-1}$ holds for all $x > 0$. Since the edge sets are chosen independently in each step and the random choices of the stations whether to transmit or not are also independent from each other we have:

$$\begin{aligned}
\mathbf{Pr}\left[\bigwedge_{i=1}^{\tau}(S_i = 0)\right] &= \prod_{i=1}^{\tau} \mathbf{Pr}[\, S_i = 0\,] \leq \prod_{i=1}^{\tau}\left(1 - \frac{q_i}{2e\Delta^{1/\tau}}\right) && \text{by Equation (2)} \\
&\leq 1 - \sum_{i=1}^{\tau} \frac{q_i}{2e\Delta^{1/\tau}} + \sum_{1 \leq i < j \leq \tau} \frac{q_i q_j}{4e^2\Delta^{2/\tau}} && \text{by Lemma 3} \\
&\leq 1 - \frac{\sum_{i=1}^{\tau} q_i}{2e\Delta^{1/\tau}} + \frac{\left(\sum_{i=1}^{\tau} q_i\right)^2}{4e^2\Delta^{2/\tau}} \\
&\leq 1 - \frac{1}{2e\Delta^{1/\tau}} + \frac{1}{4e^2\Delta^{2/\tau}} \leq 1 - \frac{1}{4e\Delta^{1/\tau}} && \text{by Equation (1)}
\end{aligned}$$

Hence if we execute the procedure for $2\tau\lceil\ln(1/\epsilon)\rceil \cdot \lceil 4e \cdot \Delta^{1/\tau}\rceil$ time steps, we have at least $\lceil\ln(1/\epsilon)\rceil \cdot \lceil 4e \cdot \Delta^{1/\tau}\rceil$ sequences of τ consecutive time steps in which the distribution over the unreliable edges is the same and the algorithm tries all the probabilities $\{p_1, p_2, \dots, p_\tau\}$. Each of these procedures fails independently with probability at most $1 - 1/(4e\Delta^{1/\tau})$ hence the probability that all the procedures fail is at most: $\left(1 - \frac{1}{4e\Delta^{1/\tau}}\right)^{\lceil\ln(1/\epsilon)\rceil \cdot \lceil 4e\Delta^{1/\tau}\rceil} \leq e^{-\lceil\ln(1/\epsilon)\rceil} < \epsilon$ ◀

On closer inspection of the analysis of Lemma 4, it becomes clear that if we tweak slightly the probabilities used in our algorithm, we require fewer iterations. In more detail, the probability of a successful transmission in the case where each of the x transmitters broadcasts independently with probability α/x is approximately $\alpha/(2e)^\alpha$. In the previous algorithm we were transmitting in successive steps with probabilities $\Delta^{-1/\tau}, \Delta^{-2/\tau}, \dots$. Thus if $x = 1$ we

would get in i-th step $\alpha = \Delta^{-i/\tau}$ and approximately the sum of probabilities of success in τ consecutive steps would be $\Delta^{-1/\tau}$. The formula $\alpha/(2e)^{-\alpha}$ shows that the success probability depends on α linearly if $\alpha < 1$ (“too small" probability) and depends exponentially on α if $\alpha > 1$ (“too large" probability). In the previous theorem we intuitively only use the linear term. In the next one we would like to also use, to some extent, the exponential term. If we shift all the probabilities by multiplying them by a factor of $\beta > 1$, the total success probability would be approximately $\beta\Delta^{-1/\tau}$ if $x = 1$ and $\beta(2e)^{-\beta}$ if $x = \Delta$. Thus by setting $\beta = \log_{2e} \Delta/\tau$ we maximize both these values.

```
1 Algorithm: FRLB(r, τ̄)
2 for i ← 1 to τ̄ do p_i ← Δ^{-i/τ̄} · log_{2e} Δ/τ̄
3 repeat r times
4   Uniform (τ̄, p_1, p_2, ..., p_τ̄)
```

The following lemma makes this above intuition precise and gains a log-factor in performance in algorithm FRLB (Fast Robust Local Broadcast) compared to RLB. As part of this analysis, we add a second statement to our lemma that will prove useful during our subsequent analysis of global broadcast. The correctness of this second lemma is a straightforward consequence of the analysis.

▶ **Lemma 5.** *Fix any receiver $u \in R$ and error bound $\epsilon > 0$. It follows:*

1. *FRLB*$(2\lceil\ln(1/\epsilon)\rceil \cdot \lceil 4\Delta^{1/\bar{\tau}}\bar{\tau}/\log_{2e}\Delta\rceil, \bar{\tau})$ *completes local broadcast with a single receiver in time* $O\left(\frac{\Delta^{1/\bar{\tau}}\cdot\bar{\tau}^2}{\log\Delta}\cdot\log(1/\epsilon)\right)$ *with probability at least $1-\epsilon$, for any $\epsilon > 0$,*
2. *FRLB*$(2, \bar{\tau})$ *completes local broadcast with a single receiver with probability at least* $\frac{\log_{2e}\Delta}{4\Delta^{1/\bar{\tau}}\bar{\tau}}$.

Proof Idea. The proof is similar to the one of Lemma 4. We define the probabilities q_i and events Q_i in the same way. The key difference is in the evaluation of the probability of success in round t_i conditioned on Q_i ($\mathbf{Pr}[\,S_i = 1 \mid Q_i\,]$). Event Q_i restricts the number of neighbors connected to u to some interval. We prove that the success probability $\mathbf{Pr}[\,S_i = 1 \mid Q_i\,]$ is lower bounded by the minimum of the values at the endpoints of this interval. This is true because when x stations transmit with probability p to a common neighbor then the probability of a successful transmission seen as a function of x has a single maximum at $x = 1/p$ hence its value at any point of some fixed interval is lower bounded by the minimum of the values at the endpoints. ◀

In Lemmas 4 and 5 we studied the fate of a single receiver in R during an execution of algorithms RLB and FRLB. Here we apply this result to bound the time for all nodes in R to receive a message, therefore solving the local broadcast problem. In particular, for a desired error bound ϵ, if we apply these lemmas with error bound $\epsilon' = \epsilon/n$, then we end up solving the single node problem with a failure probability upper bounded by ϵ/n. Applying a union bound, it follows that the probability that any node from R fails to receive a message is less than ϵ. Formally:

▶ **Theorem 6.** *Fix an error bound $\epsilon > 0$. It follows that algorithm* *FRLB*$(2\lceil\ln(n/\epsilon)\rceil \cdot \lceil 4\Delta^{1/\bar{\tau}}\bar{\tau}/\log\Delta\rceil)$ *solves local broadcast in* $O\left(\frac{\Delta^{1/\bar{\tau}}\cdot\bar{\tau}^2}{\log_{2e}\Delta}\cdot\log(n/\epsilon)\right)$ *rounds, with probability at least $1-\epsilon$.*

3.2 Lower bound

Observe that for $\tau = \Omega(\log \Delta)$, FRLB has a time complexity of $O(\log \Delta \log n)$ rounds for $\epsilon = 1/n$, which matches the performance of the optimal algorithms for this problem in the standard radio model. This emphasizes the perhaps surprising result that even large amounts of topology changes do not impede simple uniform broadcast strategies, so long as there is independence between nearby changes.

Once τ drops below $\log \Delta$, however, a significant gap opens between our model and the standard radio network model. Here we prove that gap is fundamental for any uniform algorithm in our model.

In the local broadcast problem, a receiver from set R can have between 1 and Δ neighbors in set B. The neighbors should optimally use probabilities close to the inverse of their number. But since the number of neighbors is unknown, the algorithm has to check all the values. If we look at the logarithm of the inverse of the probabilities (call them *log-estimates*) used in Lemma 4 we get $i \log \Delta/\tau$, for $i = 1, 2, \ldots, \tau$ – which are spaced equidistantly on the interval $[0, \log \Delta]$. The goal of the algorithm is to minimize the maximum gap between two adjacent log-estimates placed on this interval since this maximizes the success probability in the worst case. With this in mind, in the proof of the following lower bound, we look at the dual problem. Given a predetermined sequence of probabilities used by an arbitrary uniform algorithm, we seek the largest gap between adjacent log-estimates, and then select edge distributions that take advantage of this weakness.

▶ **Theorem 7.** *Fix a maximum degree $\Delta \geq 10$, stability factor $\tau \leq \log(\Delta-1)/16$, and uniform local broadcast algorithm $\mathcal{A}$. Assume that $\mathcal{A}$ guarantees with probability at least $1/2$ to solve local broadcast in $f(\Delta,\tau)$ rounds when executed in any dual graph network with maximum degree Δ and fading adversary with stability τ. It follows that $f(\Delta,\tau) \in \Omega(\Delta^{1/\tau}\tau/\log \Delta)$.*

Proof Idea. In this proof we use a star with Δ arms out of which only one is reliable – all other arms are controlled by the adversary. The single receiver u is the center of the star. For any uniform algorithm we divide the probabilities p_i into sequences of length τ and find a distribution in which the degree of u is "hard" for each sequence. The algorithm places τ log-estimates on interval $[0, \log \Delta]$ we, as an adversary, can clearly find a largest gap between adjacent log-estimates of length approximately $\log \Delta/\tau$. We choose the degree d of u such that its logarithm is inside this gap (in correct distances from both its endpoints). With this choice we can upper bound the probability of a successful transmission in any step during these τ steps, because the distance between the log-estimate and the logarithm of the degree of u gives us lower bound on dp_i if $p_i > 1/d$ or of $1/(dp_i)$ if $p_i < 1/d$ which in turn upper bounds the probability of a successful transmission. ◀

In our next theorem, we refine the argument used in Theorem 7 for the case where τ is a non-trivial amount smaller than the $\log \Delta$ threshold. We will argue that for smaller τ, the complexity is $\Omega(\Delta^{1/\tau}\tau^2/\log \Delta)$, which more exactly matches our best upper bound. We are able to trade this small amount of extra wiggle room in τ for a stronger lower bound because it simplifies certain probabilistic obstacles in our argument. Combined with our previous theorem, the below result shows our upper bound performance is asymptotically optimal for uniform algorithms for all but a narrow range of stability factors, for which it is near tight.

▶ **Theorem 8.** *Fix a maximum degree $\Delta \geq 10$, stability factor $\tau \leq \ln(\Delta-1)/(12 \log\log(\Delta-1))$, and uniform local broadcast algorithm $\mathcal{A}$. Assume that $\mathcal{A}$ guarantees with probability at least $1/2$ to solve local broadcast in $f(\Delta,\tau)$ rounds when executed in any dual graph network with maximum degree Δ and fading adversary with stability τ. It follows that $f(\Delta,\tau) \in \Omega(\Delta^{1/\tau}\tau^2/\log \Delta)$.*

Proof Idea. The proof is similar to proof of Theorem 7. Here we also find a gap of length $\log \Delta/\tau$ and then we argue that in a "proximity" of each such a large gap there has to exist a large number of log-estimates. The proximity is defined so that all log-estimates outside of it are (almost) irrelevant, give a very small probability of success, if we choose the logarithm of the degree of u to be inside the considered gap. This in turn implies that in the remaining part of the interval the "density" of log-estimates is lower hence there must exist another large gap. By repeating this argument we can derive a contradiction with the assumed time complexity. The reason why we need to restrict τ is that our defined proximity must be of the same order as $\log \Delta/\tau$ which is no longer true for τ being close to $\log \Delta$. ◀

4 Global Broadcast

We now turn our attention to the global broadcast problem. Our upper bound will use the same broadcast probability sequence as our best local broadcast algorithm from before. As with local broadcast, for $\tau \geq \log \Delta$, our performance nearly matches the optimal performance in the standard radio network model, and then degrades as τ shrinks toward 1. Our lower bound will establish that this degredation is near optimal for uniform algorithms in this setting. In this section we also use the notation $\bar{\tau} = \min\{\tau, \lceil \log \Delta \rceil\}$.

4.1 Upper Bound

A uniform global broadcast algorithm requires each node to cycle through a predetermined sequence of broadcast probabilities once it becomes *active* (i.e., has received the broadcast message). The only slight twist in our algorithm's presentation is that we assume that once a node becomes active, it waits until the start of the next probability cycle to start broadcasting. To implement this logic in pseudocode, we use the variable *Time* to indicate the current global round count. We detail this algorithm below (notice, the FRLB(2) is the local broadcast algorithm analyzed in Lemma 5).

```
1 Algorithm: RGB(ε)
2 Wait until receiving the message
3 Wait until (Time mod 2τ̄) = 0
4 repeat ⌈ln (2n/ε)⌉ · ⌈4Δ^{1/τ̄} τ̄/ log Δ⌉ times
5   FRLB(2)
```

▶ **Theorem 9.** *Fix an error bound* $\epsilon > 0$. *It follows that algorithm RGB*(ϵ) *completes global broadcast in time* $O\left((D + \log(n/\epsilon)) \cdot \frac{\Delta^{1/\bar{\tau}}\bar{\tau}^2}{\log \Delta}\right)$, *with probability at least* $1 - \epsilon$.

Proof Idea. Here we use the same idea as in the proof of [2, Theorem 4]. There a local broadcast algorithm (*Decay*) is used as a black box in a global broadcast algorithm. We use a different local broadcast algorithm (FRLB) but the same analysis applies. ◀

4.2 Lower Bound

The global broadcast lower bound of $\Omega(D \log(n/D))$, proved by Kushilevitz and Mansour [13] for the standard radio network model, clearly still holds in our setting, as the radio network model is a special case of the dual graph model where $E' = E$. Similarly, the $\Omega(\log n \log \Delta)$

lower bound proved by Alon *et al.* [1] also applies.[3] It follows that for $\tau \geq \log \Delta$, we almost match the optimal bound for the standard radio network model, and do match the time of the seminal algorithm of Bar-Yehuda et al. [2].

For smaller τ, this performance degrades rapidly. Here we prove this degradation is near optimal for uniform global broadcast algorithms in our model. We apply the obvious approach of breaking the problem of global broadcast into multiple sequential instances of local broadcast (though there are some non-obvious obstacles that arise in implementing this idea). As with our local broadcast lower bounds, we separate out the case where τ is at least a $1/\log\log\Delta$ factor smaller than our $\log\Delta$ threshold, as we can obtain a slightly stronger bound under this assumption.

▶ **Theorem 10.** *Fix a maximum degree $\Delta \geq 10$, stability factor τ, diameter $D \geq 24$ and uniform global broadcast algorithm $\mathcal{A}$. Assume that $\mathcal{A}$ solves global broadcast in expected time $f(\Delta, D, \tau)$ in all graphs with diameter D, maximum degree Δ and fading adversary with stability τ. It follows that:*

1. *if $\tau < \ln(\Delta-1)/(12\log\log(\Delta-1))$ then $f(\Delta, D, \tau) \in \Omega(D\Delta^{1/\tau}\tau^2/\log\Delta)$,*
2. *if $\tau < \ln(\Delta-1)/16$ then $f(\Delta, D, \tau) \in \Omega(D\Delta^{1/\tau}\tau/\log\Delta)$.*

Proof Idea. In this proof we connect together $\Omega(D)$ gadgets used in the proof of Theorem 7 (and 8) and lower bound the time the message spends in each of the gadgets. The only problem in this approach is that after the message enters to the next gadget, the adversary might not be allowed to change the distribution for some number of steps. We solve this by keeping a distribution that is "hard" for the first τ probabilities of the algorithm in each of the gadgets that has not been reached by the message yet. ◀

5 Correlations

Here we explore a promising direction for the study of broadcast in realistic radio network models. In particular, the fading adversary studied above assumes that the distribution draws are independent. As we will show, interesting results are still possible when considering the even more general case where the marginal distributions in each step are not necessarily independent in each round. More precisely, in this case, the adversary chooses a distribution over sequences of length at least τ of the sets of unreliable edges. A sequence from this distribution is used to determine which unreliable edges are active in successive steps. The adversary after a least τ steps can decide to change the distribution. In this model, we first show a simple lower bound that any uniform algorithm using a short list of probabilities of length l (our algorithms in previous sections always used list of length $\min\{\tau, \log\Delta\}$) needs time $\Omega(\sqrt{n}/l)$ for some graphs. Our lower bound uses distributions over sequences of graphs in which the degrees of nodes change by a large number in successive steps. Such large changes in degree turn out to be crucial as we show that if in the sequence taken from the distribution chosen by the adversary, in every step in expectancy only $O(\Delta^{1/(\tau-o(\tau))})$ edges adjacent to each node can be changed then we can get an algorithm working in time $O(\Delta^{1/\tau}\tau\log(1/\epsilon))$ with probability at least $1-\epsilon$ and using list of probabilities of length $O(\min\{\tau, \log\Delta\})$.

[3] This bound is actually stated as $\Omega(\log^2 n)$, but $\Delta = \Theta(n)$ in the lower bound network, so it can be expressed in terms of Δ as well for our purposes here.

5.1 A Lower Bound for Correlated Distributions

The following lower bound shows that any simple back-off algorithm, similar to the ones presented in Section 3, that uses at most $\log \Delta$ probabilities requires time $\Omega(\sqrt{\Delta}/\log \Delta)$ if arbitrary correlations are permitted.

▶ **Proposition 11.** *Any uniform local broadcast algorithm that repeats a procedure consisting of l probabilities requires expected time $\Omega(\sqrt{\Delta}/l)$ in graph with $\Delta = n - 2$ even if $\tau = \infty$.*

Proof. Denote the procedure that is being used by the algorithm by $\mathcal{P}$. Assume for simplicity that $\sqrt{\Delta}$ is a natural number. We take as a graph a connected pair of stars.

The fist star has arms $v_1, v_2, \ldots, v_\Delta$ and center at u. In the fist star, arms $v_1, v_2, \ldots, v_\Delta$ are connected to center u by reliable edges. The second star has arms $v_1, v_2, \ldots, v_\Delta$ and center at v. In the second star, connection from v_1 to v is reliable and all other connections are unreliable. Note that by such construction, graph G is connected. All nodes, except v, are initially holding a message.

The single distribution is defined in the following way. Let $e_i = \min\{1/p_i, \Delta\}$ for $i = 1, 2, \ldots, l$ be the estimates used by procedure $\mathcal{P}$. Let $\bar{e}_i = \begin{cases} 1 \text{ if } e_i \geq \sqrt{\Delta}, \\ n \text{ otherwise }. \end{cases}$ Let s be a number chosen uniformly at random from $\{1, 2, \ldots, l\}$. In our distribution, the degree of v in step t is $d_t = \bar{e}_{1+r_t}$, where r_t is the remainder of $t + s$ modulo l. More precisely, in step t in the distribution exactly $d_t - 1$ edges chosen at random among edges between v and $v_2, v_3, \ldots, v_\Delta$ are activated. Observe that before the algorithm starts, the distribution of the degree of node v in each step is simply a uniform number from multiset $\{\bar{e}_1, \bar{e}_2, \ldots, \bar{e}_l\}$. But after step 1 the sequence of degrees of v becomes deterministic and depends only on the value s of the shift. The dependencies are designed in such a way that if $s = l$ (which happens with probability $1/l$) then in any step t of the algorithm, the probability p_t used by the algorithm satisfies either $p_t \cdot d_t \geq \sqrt{\Delta}$ or $p_t \cdot d_t < 1/\sqrt{\Delta}$. This means by Lemma 1 that the success probability is at most $1/\sqrt{\Delta}$ in each step and hence by the union bound the success probability in the whole procedure is at most $l/\sqrt{\Delta}$. Thus with probability at least $1/l$ the algorithm has to repeat procedure $\mathcal{P}$ at least $\sqrt{\Delta}/(2l)$ times to get a constant probability of success. Hence the expected time is $\Omega(\sqrt{\Delta}/l)$. ◀

5.2 Locally Limited Changes

The previous section shows that under an adversary that is allowed to use arbitrary correlations then any simple procedure need polynomial time in the worst case.

In this section we want to consider the adversary that can use correlations but cannot change the degree too much in successive steps. Of course once every at most τ steps the adversary is allowed to define a completely new distribution over the unreliable edges. We want to argue that it is possible to build a simple algorithm resistant to such an adversary. Intuitively the changes of the degree are problematic only if the changes are by a large (non-constant) factor. Note by Lemma 1 that if we perturb the effective degree by only a constant factor then the bound also changes only by a constant factor. Hence in order to design an algorithm that is immune to such changes we should add more "coverage" to the small-degree nodes. We do this by enhancing each phase of algorithm `RLB` with additional steps in which we assume that the effective degree of a node is small. The adversary may try to avoid the successful transmission in these steps by changing the degree (the adversary knows the probabilities used by the algorithm). But having the restriction on the distance the adversary can move the degree allows us to define overlapping "zones" such that in two

consecutive steps we are sure to find the degree in one of the zones. We also have to make sure that the whole phase of the new algorithm fits into τ steps.

Now we present algorithm RLBC (Robust Local Broadcast with Correlations). We first show that the algorithm works under (l,τ)–deterministic adversary that can change at most l edges adjacent to each node per round and all the edges from $E' \setminus E$ once every at most τ rounds. Our algorithm will be resistant to deterministic adversary that can change at most $\tau\Delta^{1/(\tau-o(\tau))}$ edges adjacent to each node in every step.

Then we show that it also works under restricted fading adversary with parameters τ and l. Restricted fading adversary can change the distribution arbitrarily once every at most τ steps, if the distribution is not changed then the expected change of the degree of any node can be at most l. Under these restrictions, the adversary can design arbitrary correlations between successive steps. We show that RLBC works with restricted fading adversary with l of at most $\Delta^{1/(\tau-o(\tau))}$.

```
1 Algorithm: RLBC(r, τ)
2 τ̄ ← min{⌈log_{2e} Δ/2⌉, τ}; a ← ⌈τ̄/log_{2e} τ̄⌉; k ← ⌈Δ^{1/(τ−2a)}⌉
3 e1 ← k · a; e2 ← k^2 · τ · a
4 repeat 2r times
5     RLB(1, τ̄ − 2a)
6     repeat a times
7         Uniform (1, 1/e1)
8         Uniform (1, 1/e2)
```

▶ **Theorem 12.** *If $\tau \geq 1000$ Algorithm RLBC$(8e\lceil\ln(1/\epsilon)\Delta^{1/\tau}\rceil,\tau)$ solves local broadcast in the presence of $\left(\left\lfloor\Delta^{\frac{1}{\tau-2\lceil\tau/\log_{2e}\tau\rceil}}\right\rfloor\tau/2,\tau\right)$-deterministic adversary in time $O(\Delta^{1/\tau}\tau\log(1/\epsilon))$ with probability at least $1-\epsilon$.*

Proof Idea. For a fixed receiver v we want to show that the probability that v receives the message in one of the r cycles (each 2 iterations of loop in Lines $7-11$ is one cycle) is at least $p_s = \frac{1}{8ek}$. We do it by separately considering two cases depending on degree $d_t(v)$, where t is the first step of the considered cycle. If $d_t(v) \geq 2l^2$ we can show that the degree cannot change in total in this cycle by more than a factor of 2 (here we use the restriction on the adversary) in which case we can show that in one of the steps of procedure RLB the probability of success is at least p_s. For smaller degrees $d_t(v) < 2l^2$ we pick a pairs of steps such that in the first step of the pair the algorithm uses probability $1/e_1$ and in the second it uses $1/e_2$. Then we observe that either in the first step of the pair the degree is at most $2l$ in which case broadcasting with probability $1/e_1$ gives probability p_s/a of success. In the opposite case the degree is at least l (here we use the restriction on the adversary) in the second step and broadcasting with probability $1/e_2$ gives probability p_s/a of success. Since we have a such pairs the claim follows. ◀

The case with deterministic adversary can be generalized to stochastic restricted adversary.

▶ **Theorem 13.** *If $\tau \geq 1000$ Algorithm RLBC$(16e\lceil\ln(1/\epsilon)\Delta^{1/\tau}\rceil,\tau)$ solves local broadcast in the presence of l-restricted fading adversary using correlations with $l = \left\lfloor\Delta^{\frac{1}{\tau(1-1/\log_{2e}\tau)}}\right\rfloor/4$ in time $O(\Delta^{1/\tau}\tau\log(1/\epsilon))$ with probability at least $1-\epsilon$.*

Proof Idea. We show that if an algorithm works with $2l\tau$-deterministic adversary then it also works with l-stochastic adversary with correlations. We note that by Markov's inequality with probability at least $1/(2\tau)$ the degree of the receiver changes by at most $2l\tau$. By the union bound with probability at least $1/2$, the degree does not change by more then $2l\tau$ throughout the whole cycle of length τ. For such cycles, the analysis of the deterministic case gives us probability p_s of success. Thus in the stochastic case the probability of success in each cycle is at least $p_s/2$. ◀

References

1. N. Alon, A. Bar-Noy, N. Linial, and D. Peleg. A Lower Bound for Radio Broadcast. *Journal of Computer and System Sciences*, 43(2):290–298, 1991.
2. Reuven Bar-Yehuda, Oded Goldreich, and Alon Itai. On the Time-Complexity of Broadcast in Multi-hop Radio Networks: An Exponential Gap Between Determinism and Randomization. *J. Comput. Syst. Sci.*, 45(1):104–126, 1992. doi:10.1016/0022-0000(92)90042-H.
3. Keren Censor-Hillel, Seth Gilbert, Fabian Kuhn, Nancy Lynch, and Calvin Newport. Structuring Unreliable Radio Networks. *Distributed Computing*, 27(1):1–19, 2014.
4. Andrea E. F. Clementi, Angelo Monti, and Riccardo Silvestri. Round Robin is optimal for fault-tolerant broadcasting on wireless networks. *J. Parallel Distrib. Comput.*, 64(1):89–96, 2004. doi:10.1016/j.jpdc.2003.09.002.
5. Mohsen Ghaffari. Bounds on Contention Management in Radio Networks. Master's thesis, Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, February 2013.
6. Mohsen Ghaffari, Bernhard Haeupler, Nancy Lynch, and Calvin Newport. Bounds on Contention Management in Radio Networks. In Marcos K. Aguilera, editor, *Distributed Computing: 26th International Symposium (DISC 2012), Salvador, Brazil, October, 2012*, volume 7611 of *Lecture Notes in Computer Science*, pages 223–237. Springer, 2012.
7. Mohsen Ghaffari, Erez Kantor, Nancy Lynch, and Calvin Newport. Multi-Message Broadcast with Abstract MAC Layers and Unreliable Links. In *Proceedings of the 33nd Annual ACM Symposium on Principles of Distributed Computing (PODC'14)*, pages 56–65, Paris, France, July 2014.
8. Mohsen Ghaffari, Nancy Lynch, and Calvin Newport. The Cost of Radio Network Broadcast for Different Models of Unreliable Links. In *Proceedings of the 32nd Annual ACM Symposium on Principles of Distributed Computing*, pages 345–354, Montreal, Canada, July 2013.
9. Mohsen Ghaffari and Calvin Newport. A Leader Election in Unreliable Radio Networks. In *Proceedings of the International Colloquium on Automata, Languages, and Programming (ICALP)*, 2016.
10. Seth Gilbert, Nancy A. Lynch, Calvin Newport, and Dominik Pajak. Brief Announcement: On Simple Back-Off in Unreliable Radio Networks. In *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*, pages 48:1–48:3, 2018. doi:10.4230/LIPIcs.DISC.2018.48.
11. Fabian Kuhn, Nancy Lynch, and Calvin Newport. Brief Announcement: Hardness of Broadcasting in Wireless Networks with Unreliable Communication. In *Proceedings of the 28th Annual ACM Symposium on the Principles of Distributed Computing (PODC 2009)*, Calgary, Alberta, Canada, August 2009.
12. Fabian Kuhn, Nancy Lynch, Calvin Newport, Rotem Oshman, and Andrea Richa. Broadcasting in Unreliable Radio Networks. In *Proceedings of the 29th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 336–345, Zurich, Switzerland, July 2010.

13 E. Kushilevitz and Y. Mansour. An $\Omega(D \log(N/D))$ Lower Bound for Broadcast in Radio Networks. *SIAM Journal on Computing*, 27(3):702–712, 1998.

14 Nancy Lynch and Calvin Newport. A (Truly) Local Broadcast Layer for Unreliable Radio Networks. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 2015.

15 Calvin Newport. Lower Bounds for Radio Networks Made Easy. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, 2014.

16 Mike Willis. Propagation Tutorial: Fading. `http://www.mike-willis.com/Tutorial/PF15.htm`. May 5, 2007.

Concurrent Specifications Beyond Linearizability

Éric Goubault
École Polytechnique, Palaiseau, France
eric.goubault@lix.polytechnique.fr

Jérémy Ledent
École Polytechnique, Palaiseau, France
jeremy.ledent@lix.polytechnique.fr

Samuel Mimram
École Polytechnique, Palaiseau, France
samuel.mimram@lix.polytechnique.fr

Abstract

With the advent of parallel architectures, distributed programs are used intensively and the question of how to formally specify the behaviors expected from such programs becomes crucial. A very general way to specify concurrent objects is to simply give the set of all the execution traces that we consider correct for the object. In many cases, one is only interested in studying a subclass of these concurrent specifications, and more convenient tools such as linearizability can be used to describe them.

In this paper, what we call a concurrent specification will be a set of execution traces that moreover satisfies a number of axioms. As we argue, these are actually the only concurrent specifications of interest: we prove that, in a reasonable computational model, every program satisfies all of our axioms. Restricting to this class of concurrent specifications allows us to formally relate our concurrent specifications with the ones obtained by linearizability, as well as its more recent variants (set- and interval-linearizability).

2012 ACM Subject Classification Theory of computation → Concurrency

Keywords and phrases concurrent specification, concurrent object, linearizability

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.28

Eligible for best student paper award

1 Introduction

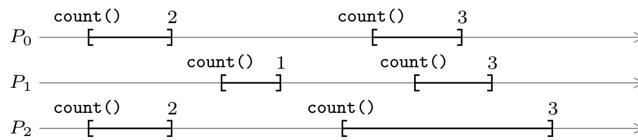
A common setting to study distributed computing is the one of asynchronous processes communicating through shared objects. It is of particular interest in the area of fault-tolerant distributed computing [9], where we assume that some processes might crash during the computation. The goal is to determine which concurrent tasks are solvable by wait-free programs [8], depending on which shared objects the processes are allowed to use. In this context, the question of how to formally specify the behavior of the shared objects arises: what we want is an abstract, high-level specification, that does not refer to a particular implementation of the object. To illustrate our discussion, let us first introduce a toy (and running) example which we call the COUNT object.

© Éric Goubault, Jérémy Ledent, and Samuel Mimram;
licensed under Creative Commons License CC-BY
22nd International Conference on Principles of Distributed Systems (OPODIS 2018).
Editors: Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira; Article No. 28; pp. 28:1–28:16

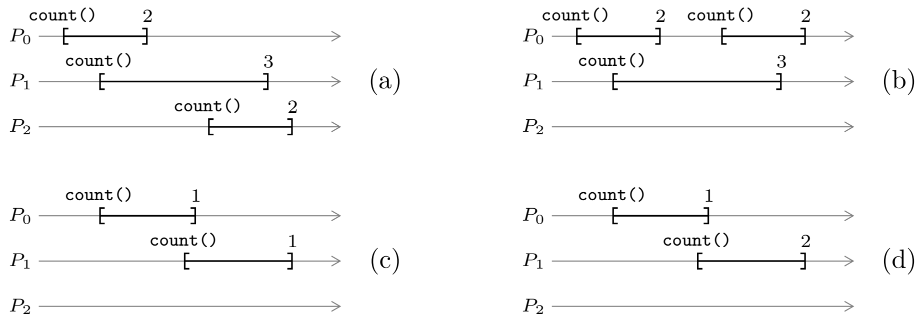
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1.1 Counting processes

When a process calls the COUNT object, it should return the number of processes that are currently calling the object in parallel. This is similar, although not completely identical, to Java's `Thread.activeCount` method, which returns "an estimate of" the current number of running threads. A typical execution of this object is depicted below. Each of the three processes P_0, P_1 and P_2 is calling the COUNT object twice. The horizontal axis represents the real-time scheduling of the operations and the intervals between square brackets depict the time span during which a process is executing the `count()` method: when two intervals vertically overlap, the processes are running the method concurrently. For instance, the last three calls are concurrent: the three processes should see each other and all return 3.

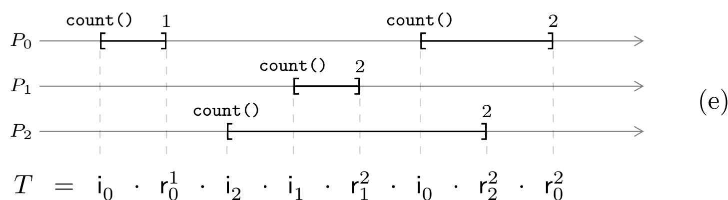


The specification of COUNT seems to be clear on this example. But what about the behaviors depicted below? In execution (a), process P_1 responds 3 since it has seen the two other processes; but there is no point in time when the three processes are running concurrently. Execution (b) represents the same situation, but this time P_1 has seen two different calls of P_0. In execution (c), the two calls are concurrent, but for a very short time, so they did not manage to see each other. In execution (d), process P_1 managed to see process P_0, but not reciprocally. All of these executions may or may not seem correct depending on what exact specification we have in mind. One could for example ask for a variant of the COUNT object that accepts executions (a) and (c), but rejects (b) and (d), and so on.



1.2 Specifying concurrent objects

A simple but very general way of specifying such objects was proposed by Lamport [13]. The specification of a concurrent object is simply the set of all the execution traces that we consider correct for this object. For the COUNT object example, if we draw every possible diagram as in the pictures above and decide which ones we want or not, we will have a well-specified object. Of course, we need a mathematical abstraction of these pictures, we represent them by *execution traces* (Lamport originally used the happens-before partial order; the trace formalism used here was introduced by Herlihy and Wing [10]).



(e)

An implicit assumption of this representation is that invocations and responses are totally ordered according to some global time clock. Another assumption is that only the relative position of the events in time matters: the intervals can be moved around as long as the overlapping pattern is conserved. In particular, we cannot tell how long processes overlap, so execution (c) is indistinguishable from an execution where P_0 and P_1 are fully concurrent.

Although very powerful in terms of expressiveness, this idea of specifying a concurrent object by giving the set of all correct executions can be cumbersome to work with in practice.

Very often, one is only interested in studying concurrent objects that have a "sequential flavor", i.e., objects that are concurrent versions of sequential data structures, such as lists or queues. In this scenario, it is usually better to use one of the many correctness criteria found in the literature such as atomicity [16], sequential consistency [12], serializability [19], causal consistency [20], or linearizability [10]. In fact, those correctness criteria can be regarded as convenient ways of defining concurrent specifications in the sense of Lamport. For example, given the sequential specification σ of some object (say, a list), we can generate the set $\mathsf{Lin}(\sigma)$ of *linearizable* concurrent traces, that is, the set of correct executions of a concurrent list (according to linearizability). Thus, $\mathsf{Lin}(\sigma)$ is a concurrent specification (in the sense of Lamport) which is parameterized by a sequential specification σ. Note that σ is a much better-understood mathematical object than general concurrent specifications.

The drawback of these methods is that they cannot specify every concurrent object. The original paper on linearizability [10] already remarked that Lamport's approach is more general, since it can express non-linearizable behavior. The COUNT object described above is a typical example of an object that exhibits intrinsically concurrent behavior, which cannot be specified by sequential means. Many other examples are found in the area of distributed computability [9], such as immediate snapshot, consensus or set-agreement objects. Another notable example is Java's *Exchanger* object. In order to specify those objects, many variants of linearizability have been defined recently: local linearizability [6], set-linearizability [17] (a.k.a. concurrency-aware linearizability [7]) and interval-linearizability [2]. The latter notion is the most expressive one; in particular, they show that their framework allows to specify every concurrent *task*, in the sense of distributed computability [9].

1.3 Contributions

In this paper, we define a notion of *concurrent specification* which is based on Lamport's idea of a set of correct execution traces; but we moreover impose a number of axioms that should hold on this set. We advocate the relevance of this definition as follows.

Firstly, we define a reasonable computational model (thought of as the operational semantics of a concurrent programming language with shared objects), and prove that in this model, every program satisfies our axioms (Theorem 12). This means that our concurrent specifications are the only relevant ones: a specification that does not verify our axioms could never be implemented by a program. Thus, imposing those axioms is not very restrictive; rather, they are desirable properties that concurrent specifications should satisfy.

Secondly, this particular set of axioms allows us to relate formally our specifications and the linearizability-based ones, by constructing Galois connections between them (Theorem 16). In particular, as a corollary, we obtain that our concurrent specifications coincide with the ones that are definable using interval-linearizability. Beyond the applications that we show here, this theorem is also interesting in itself: it states formally in what sense linearizability is the canonical way to turn a sequential specification into a concurrent one. Indeed, given a sequential specification σ, linearizability gives the smallest possible concurrent specification, among the ones that both contain σ and satisfy our axioms.

1.4 Plan

We begin by introducing our definition of concurrent specification (Section 2). Then, we provide an operational model that validates our axioms (Section 3), and we establish Galois connections with various notions of linearizable specification (Section 4). Finally, we conclude (Section 5).

2 Concurrent specifications

In the rest of the paper, we suppose fixed a number $n \in \mathbb{N}$ of *processes* and write $[n] = \{0, 1, \dots, n-1\}$ for the set of *process names* (a process is identified by its number). We also suppose fixed a set $\mathcal{V}$ of *values* which can be exchanged between processes and objects (typically $\mathcal{V} = \mathbb{N}$). The set $\mathcal{A}$ of possible *actions* for an object is defined as

$$\mathcal{A} \quad = \quad \{\mathsf{i}_i^x \mid i \in [n], x \in \mathcal{V}\} \cup \{\mathsf{r}_i^y \mid i \in [n], y \in \mathcal{V}\}$$

An action is thus either

- i_i^x: an *invocation* of the object by the process i with input value x,
- r_i^y: a *response* of the object to the process i with output value y.

An *execution trace* is a finite sequence of actions, and we write $\mathcal{T} = \mathcal{A}^*$ for the set of those; we also write ε for the empty trace and $T \cdot T'$ for the concatenation of two traces T and T'.

Given a process $i \in [n]$, the i-th *projection* $\pi_i(T)$ of a trace $T \in \mathcal{T}$ is the trace obtained from T by removing all the actions of the form i_j^x or r_j^x with $j \neq i$. A trace $T \in \mathcal{T}$ is *alternating* if for all $i \in [n]$, $\pi_i(T)$ is either empty or it begins with an invocation and alternates between invocations and responses, i.e., using the traditional notation of regular expressions:

$$\pi_i(T) \quad \in \quad \Big(\bigcup_{x,y \in \mathcal{V}} \mathsf{i}_i^x \cdot \mathsf{r}_i^y\Big)^* \cdot \Big(\bigcup_{x \in \mathcal{V}} \mathsf{i}_i^x + \varepsilon\Big)$$

In the remaining of the paper, we will only consider alternating traces. If $\pi_i(T)$ ends with an invocation, we call it a *pending* invocation. An alternating trace T is *complete* if it does not have any pending invocation.

▶ **Definition 1.** A *concurrent specification* σ is a subset of $\mathcal{T}$ which is

(1) *alternating*: every $T \in \sigma$ is alternating,
(2) *prefix-closed*: if $T \cdot T' \in \sigma$ then $T \in \sigma$,
(3) *non-empty*: $\varepsilon \in \sigma$,
(4) *receptive*: if $T \in \sigma$ and $\pi_i(T)$ has no pending invocation, then $T \cdot \mathsf{i}_i^x \in \sigma$ for every $x \in \mathcal{V}$,
(5) *total*: if $T \in \sigma$ and $\pi_i(T)$ has a pending invocation, there exists $x \in \mathcal{V}$ such that $T \cdot \mathsf{r}_i^x \in \sigma$,
(6) has *commuting invocations*: if $T \cdot \mathsf{i}_i^x \cdot \mathsf{i}_j^y \cdot T' \in \sigma$ then $T \cdot \mathsf{i}_j^y \cdot \mathsf{i}_i^x \cdot T' \in \sigma$,

(7) has *commuting responses*: if $T \cdot \mathsf{r}_i^x \cdot \mathsf{r}_j^y \cdot T' \in \sigma$ then $T \cdot \mathsf{r}_j^y \cdot \mathsf{r}_i^x \cdot T' \in \sigma$,
(8) is closed under *expansion*: if $T \cdot \mathsf{r}_j^y \cdot \mathsf{i}_i^x \cdot T' \in \sigma$ with $i \neq j$ then $T \cdot \mathsf{i}_i^x \cdot \mathsf{r}_j^y \cdot T' \in \sigma$.
We write CSpec for the set of concurrent specifications.

A concurrent specification σ is the set of all executions that we consider acceptable: a program *implements* the specification σ if all the execution traces that it generates belong to σ (this will be detailed in Section 3). The axioms (1-4) are quite natural and commonly considered in the literature. They can be read as follows: an object gives answers to invocations of processes and a process has to wait for the response before invoking again the object (1), an object can do one action at a time (2), an object can do nothing (3), invocations of objects are always possible (4). The axiom (5) states that objects always answer and in a non-blocking way; this is a less fundamental axiom and more of a design choice, since we want to model wait-free computation. Note that receptivity (4) does not force objects to accept all inputs: we could have a distinguished "error" value which is returned in case of an invalid input. Similarly, an object with several inputs, or several interacting methods (for example, a stack) can be modeled by choosing a suitable set of values $\mathcal{V}$.

The conditions (6), (7) and (8) might seem more surprising, and will be crucial to establish the Galois connection of Section 4. For example, one could expect to specify an object whose behavior depends on which process was invoked first; but that would break condition (6). As we show in Section 3, in an asynchronous model, no program can implement such a specification. Let us explain intuitively why the "closure under expansion" condition holds. Suppose $T \cdot \mathsf{r}_j^y \cdot \mathsf{i}_i^x \cdot T'$ is an acceptable execution of the object we are specifying. Then, in the trace $T \cdot \mathsf{i}_i^x \cdot \mathsf{r}_j^y \cdot T'$, where process i invokes its operation a little bit earlier, i might be idle for a while and only start computing after j has returned, resulting in the same behavior. Thus, the execution $T \cdot \mathsf{i}_i^x \cdot \mathsf{r}_j^y \cdot T'$ should also be considered acceptable. Alternatively, we can also think of it as process j being idle for a while before it returns. Applied to the COUNT object example, the expansion property implies that execution (c) *must* be accepted, since it is obtained by expanding a correct sequential execution. This expansion condition reflects the idea that invocations and responses do not correspond to actual actions taken by the processes, but rather define an interval in which they are allowed to take steps.

The conditions (6) and (7) above ensure that two invocations (resp. two responses) "commute": we say that two alternating traces $T, T' \in \mathcal{T}$ are *equivalent*, written $T \equiv T'$, if one is obtained from the other by reordering the actions within each block of consecutive invocations or consecutive responses. Generally, in the rest of the paper, we are only interested in studying traces up to equivalence.

It will prove quite useful to consider operations in traces, which are pairs consisting of an invocation and its matching response. Formally,

▶ **Definition 2.** Consider an alternating trace $T = T_0 \cdots T_{k-1}$. An *operation* of process i in T is either

- a *complete operation*: a pair (p, q) such that $T_p = \mathsf{i}_i^x$ and $T_q = \mathsf{r}_i^y$ and T_q comes right after T_p in $\pi_i(T)$, or
- a *pending operation*: a pair $(p, +\infty)$ where T_p is a pending invocation,

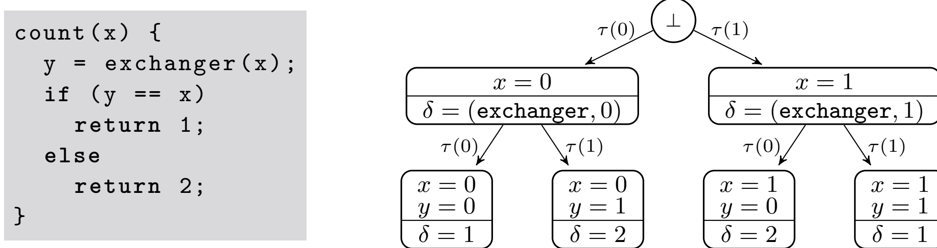
where $p, q \in \mathbb{N}$, with $0 \leq p, q < k$, are indices of actions in the trace. We write $\mathrm{op}_i(T)$ for the set of operations of the i-th process and $\mathrm{op}(T)$ for the set of all operations.

The operations of a trace can be ordered by the smallest partial order $\preceq$ such that $(p, q) \prec (p', q')$ whenever $q < p'$. This partial order is called *precedence* and two incomparable operations are called *overlapping* or *concurrent*. Note that for every $i \in [n]$, $(\mathrm{op}_i(T), \preceq)$ is totally ordered.

3 A computational model

In this section, we provide an operational model for concurrent programs communicating through shared objects. The model itself is similar to other trace-based operational semantics that can be found in the literature [4]. We assume given a set $\mathcal{O}$ of objects: they might be, for instance, concurrent data structures that have already been implemented, and that our programs are able to use in order to compute and communicate. We do not want to depend on a particular implementation of these objects, but on their specification. Thus, each object comes with its concurrent specification (as in Definition 1), which is the set of behaviors that it might exhibit. Note that our model does not have any special construct for reading and writing in the shared memory: we assume that the memory itself is given as an object in $\mathcal{O}$, with an appropriate specification. Thus, the only meaningful action a program can take is to call one of the objects; and possibly do some local computation to determine what the next call should be.

To abstract away the syntax of the programming language, we use an automata-like representation, which roughly corresponds to the control-flow graph of the program. In a given state of the automaton, the decision function δ indicates which is the next object that the program will call, and the transition function τ says what the next state will be depending on the return value of the call. The example below shows an implementation of the COUNT object for two processes communicating using a wait-free variant of Java's exchanger object: either two processes call the exchanger concurrently and swap their values, or the process fails after some timeout and gets back its own value.



We fix a set $\mathcal{O}$ of objects, along with their concurrent specification $\mathsf{spec}(o) \in \mathsf{CSpec}$ for each $o \in \mathcal{O}$. Here, a program (i.e., the implementation of an object) is basically a piece of code which takes a value as input, makes several calls to the objects in $\mathcal{O}$ (using algebraic operations to combine their results) and finally returns a value. Formally,

▶ **Definition 3.** A *program* is given by:

- a (possibly infinite) set Q of *local states* containing an *idle state* $\bot \in Q$,
- a *decision function* $\delta : Q \setminus \{\bot\} \to (\mathcal{O} \times \mathcal{V}) \sqcup \mathcal{V}$,
- a *transition function* $\tau : Q \times \mathcal{V} \to Q \setminus \{\bot\}$.

The idle state is the one where the program is waiting to be called with some input value x, in which case it will go to state $\tau(\bot, x)$. After that, the decision function gives the next step of the process depending on the current state: either call some object with some input value, or terminate and output some value. In the case where an object is called, the transition function gives the new local state of the process, depending on the previous state and the value returned by the object.

A *protocol* P is given by a program $P_i = (Q_i, \bot_i, \delta_i, \tau_i)$ for each process $i \in [n]$. The *global state* of a protocol P is an element $q = (q_0, \ldots, q_{n-1})$ of $\overline{Q} = \prod_i Q_i$, consisting of a

state for each process P_i. The initial state is $q_{\text{init}} = (\bot_0, \dots, \bot_{n-1})$. We now describe the set $\mathcal{A}$ of possible actions for P, as well as their effect $\Delta : \overline{Q} \times \mathcal{A} \to \overline{Q}$ on global states.

i_i^x: the i-th process is called with input value $x \in \mathcal{V}$. The local state q_i of process i is changed from $\bot_i$ to $\tau_i(\bot_i, x)$:

$$\Delta(q, \mathsf{i}_i^x) \quad = \quad q[i \leftarrow \tau_i(\bot_i, x)] \tag{9}$$

where the state on the right is q where q_i has been replaced by $\tau_i(\bot_i, x)$.

$\mathsf{i}(o)_i^x$: the i-th process invokes the object $o \in \mathcal{O}$ with input value $x \in \mathcal{V}$. This does not have any effect on the global state:

$$\Delta(q, \mathsf{i}(o)_i^x) \quad = \quad q \tag{10}$$

$\mathsf{r}(o)_i^x$: the object $o \in \mathcal{O}$ returns some output value $x \in \mathcal{V}$ to the i-th process. The local state of process i is updated according to its transition function τ_i:

$$\Delta(q, \mathsf{r}(o)_i^x) \quad = \quad q[i \leftarrow \tau_i(q_i, x)] \tag{11}$$

r_i^x: the i-th process has finished computing, returning the output value $x \in \mathcal{V}$. It goes back to idle state:

$$\Delta(q, \mathsf{r}_i^x) \quad = \quad q[i \leftarrow \bot_i] \tag{12}$$

The actions of the form i_i^x and r_i^x (resp. $\mathsf{i}(o)_i^x$ and $\mathsf{r}(o)_i^x$) are called *outer* (resp. *inner*) *actions*. Given a trace $T \in \mathcal{A}^*$ and an object o, we denote by T_o, called the *inner projection on* o, the trace obtained from T by keeping only the inner actions of the form $\mathsf{i}(o)_i^x$ or $\mathsf{r}(o)_i^y$. The function Δ is extended as expected as a function $\Delta : \overline{Q} \times \mathcal{A}^* \to \overline{Q}$, i.e., $\Delta(q, T \cdot T') = \Delta(\Delta(q, T), T')$ and so on. A trace is valid if at each step in the execution, the next action is taken according to the decision function δ. Formally:

▶ **Definition 4.** A trace $T \in \mathcal{A}^*$ is *valid* when for every strict prefix U of T, writing $T = U \cdot a \cdot V$ and $q' = \Delta(q_{\text{init}}, U)$, with $a \in \mathcal{A}$, we have:

- either $a = \mathsf{i}_i^x$ and $q'_i = \bot_i$;
- or $a = \mathsf{r}_i^y$ and $q'_i \neq \bot_i$ and $\delta_i(q'_i) = y$;
- or $a = \mathsf{i}(o)_i^x$ and $q'_i \neq \bot_i$ and $\delta_i(q'_i) = (o, x)$;
- or $a = \mathsf{r}(o)_i^y$ and $q'_i \neq \bot_i$.

Moreover, we require that for every object $o \in \mathcal{O}$, the inner projection T_o belongs to $\mathsf{spec}(o)$. The set of valid traces for P is written $\mathcal{T}_P \subseteq \mathcal{A}^*$.

A protocol P is *wait-free* if there is no valid infinite trace (i.e., all its prefixes are valid) involving only inner-i-actions after some position. Given a trace $T \in \mathcal{A}^*$, we write $\pi(T)$ for the trace obtained by keeping only outer actions.

▶ **Definition 5.** The *semantics* of a protocol P is the set of traces $[\![P]\!] = \{\pi(T) \mid T \in \mathcal{T}_P\}$ and P *implements* a concurrent specification σ whenever $[\![P]\!] \subseteq \sigma$, i.e., all its outer traces are valid with respect to σ.

We will now show that if P is wait-free, then $[\![P]\!]$ itself is a concurrent specification (Theorem 12). This means that in our model, the set of traces produced by a program necessarily satisfies all the axioms of concurrent specifications. The wait-free assumption is only used to prove totality; all the other axioms are true for any program. In the following, we use uppercase letters T, T' to denote traces containing only outer actions, and lowercase letters w, w', r, s, t to denote traces that might contain both inner and outer actions.

▶ **Lemma 6.** *The following commutativity properties hold:*

- commutativity of invocations*: for all* $T, T' \in \mathcal{A}^*$*,* $i, j \in [n]$ *and* $x, y \in \mathcal{V}$*,*

$$T \cdot \mathsf{i}_i^x \cdot \mathsf{i}_j^y \cdot T' \in [\![P]\!] \qquad \textit{implies} \qquad T \cdot \mathsf{i}_j^y \cdot \mathsf{i}_i^x \cdot T' \in [\![P]\!],$$

- commutativity of responses*: for all* $T, T' \in \mathcal{A}^*$*,* $i, j \in [n]$ *and* $x, y \in \mathcal{V}$*,*

$$T \cdot \mathsf{r}_i^x \cdot \mathsf{r}_j^y \cdot T' \in [\![P]\!] \qquad \textit{implies} \qquad T \cdot \mathsf{r}_j^y \cdot \mathsf{r}_i^x \cdot T' \in [\![P]\!].$$

Proof. Assume there is a word $w \in \mathcal{T}_P$ such that $\pi(w) = T \cdot \mathsf{i}_i^x \cdot \mathsf{i}_j^y \cdot T'$. So, w is of the form $r \cdot \mathsf{i}_i^x \cdot s \cdot \mathsf{i}_j^y \cdot t$ where s does not contain outer actions. We will show that $w' = r \cdot \mathsf{i}_j^y \cdot \mathsf{i}_i^x \cdot s \cdot t \in \mathcal{T}_P$, and the result follows. Since the inner projections are the same, the specifications of the objects are satisfied. We have to show that the conditions regarding the decision functions are respected. Write $q = \Delta(q_{\text{init}}, r)$ and $q' = \Delta(q_{\text{init}}, r \cdot \mathsf{i}_i^x \cdot s)$. Then since w is valid, we have $q_i = \bot_i$ and $q'_j = \bot_j$.

Claim 1: s does not contain any action of process j. Indeed, only an outer action r_j^y can set j's local state to $\bot_j$, and s has no outer action: so j's local state is $\bot_j$ during all of s. But inner actions can only be valid when the local state is not $\bot_j$.

Claim 2: Let u be a prefix of s and $q'' = \Delta(q_{\text{init}}, r \cdot \mathsf{i}_i^x \cdot u) = \Delta(q, \mathsf{i}_i^x \cdot u)$. Then $\Delta(q, \mathsf{i}_j^y \cdot \mathsf{i}_i^x \cdot u) = q''[j \leftarrow \tau_j(\bot_j, y)]$. This is proved by induction on the length of the prefix u and using Claim 1.

Claim 3: $\Delta(q, \mathsf{i}_i^x \cdot s \cdot \mathsf{i}_j^y) = \Delta(q, \mathsf{i}_j^y \cdot \mathsf{i}_i^x \cdot s)$. Take $u = s$ in Claim 2.

Finally, let u is a prefix of $w' = r \cdot \mathsf{i}_j^y \cdot \mathsf{i}_i^x \cdot s \cdot t$. If u ends in r or (by Claim 3) in t, the global state after executing it is the same as for w, so the validity condition is verified. If u ends in s, then by Claim 2 the global state only differs by its j component, but since by Claim 1 the next action is not from j, the validity condition is also verified.

For commutativity of responses, the situation is very similar: suppose that $w = r \cdot \mathsf{r}_i^x \cdot s \cdot \mathsf{r}_j^y \cdot t \in \mathcal{T}_P$, and show that $w' = r \cdot s \cdot \mathsf{r}_j^y \cdot \mathsf{r}_i^x \cdot t \in \mathcal{T}_P$. The analogue of Claim 1 says that there is no action of process i in s. ◀

▶ **Lemma 7.** $[\![P]\!]$ *has the expansion property.*

Proof. The proof is again very similar to that of Lemma 6. Assume that there is a word $w \in \mathcal{T}_P$ such that $\pi(w) = T \cdot \mathsf{r}_j^y \cdot \mathsf{i}_i^x \cdot T'$. Thus, w is of the form $r \cdot \mathsf{r}_j^y \cdot s \cdot \mathsf{i}_i^x \cdot t$, where s does not contain outer actions.

Then we have two ways of making the invocation and response commute:

- either show that $w' = r \cdot \mathsf{i}_i^x \cdot \mathsf{r}_j^y \cdot s \cdot t \in \mathcal{T}_P$, as in the proof of the commutativity of invocations,
- or show that $w'' = r \cdot s \cdot \mathsf{i}_i^x \cdot \mathsf{r}_j^y \cdot t \in \mathcal{T}_P$, as in the proof of the commutativity of responses.

These two possible proofs correspond to the two ways that we can view the expansion property that were explained in Section 2: either process i is invoked sooner and idles for a while (first proof), or process j idles before returning (second proof). ◀

The key reason that makes Lemmas 6 and 7 go through is the fact that the actions i_i^x and r_i^y do not have any effect besides starting a process or terminating it. Taking these steps does not communicate any information to the other processes. For example, a process might start running and then wait for a while before doing any "real" computation. Such a process cannot be seen by the others. This is an arbitrary choice in how we designed our computational model, but it reflects what really happens in practice: calling a function,

or returning a value, both consume some amount of clock cycles from the processor, and they do not usually have any effect on the shared memory. Thus, one could possibly have a process which has started running, but was immediately preempted by the scheduler before it could perform any meaningful operation.

▶ **Lemma 8.** *$[\![P]\!]$ is closed under prefix.*

Proof. $\mathcal{T}_P$ is closed under prefix since for all $o \in \mathcal{O}$, $\mathsf{spec}(o)$ is closed under prefix by definition, and the conditions on the decision functions are also preserved. Then, $[\![P]\!]$ is also closed under prefix. ◀

▶ **Lemma 9.** *Every trace $T \in [\![P]\!]$ is alternating.*

Proof. T must begin by an invocation because ${q_{\text{init}}}_i = \bot_i$ and the only i-action that can occur with local state $\bot_i$ is i_i^x. We then prove by induction on the length of $w \in \mathcal{A}^*$ that $\Delta(q_{\text{init}}, w)_i \neq \bot_i$ if the last outer i-action in w was an invocation, and $\Delta(q_{\text{init}}, w)_i = \bot_i$ if it was a response. This implies that no two invocations nor two responses by the same process can occur consecutively in a valid trace. ◀

▶ **Lemma 10.** *$[\![P]\!]$ is receptive.*

Proof. Assume $T \in [\![P]\!]$ where $\pi_i(T)$ does not have a pending invocation. Let $w \in \mathcal{T}_P$ such that $\pi(w) = T$. The last outer i-action in w is a response, and thus we have $\tau(q_{\text{init}}, w)_i = \bot_i$ (cf. proof of Lemma 9). So $w \cdot \mathsf{i}_i^x$ is valid for all x, and $T \cdot \mathsf{i}_i^x \in [\![P]\!]$. ◀

▶ **Lemma 11.** *$[\![P]\!]$ is total.*

Proof. Assume $T \in [\![P]\!]$ where $\pi_i(T)$ has a pending invocation. Let $w \in \mathcal{T}_P$ such that $\pi(w) = T$. The last outer i-action in w is an invocation, so the local state of i after executing w is $q_i \neq \bot_i$. If $\delta_i(q_i) = y \in \mathcal{V}$, then $w \cdot \mathsf{r}_i^y$ is a valid execution, which concludes the proof. Otherwise, $\delta_i(q_i) = (o, x)$. Then $w \cdot \mathsf{i}(o)_i^x$ is valid because the object o is receptive. And since o is total, there exists y such that $w \cdot \mathsf{i}(o)_i^x \cdot \mathsf{r}(o)_i^y$ is valid. The new local state of i after executing this trace is $q'_i \neq \bot_i$, so we can iterate the previous reasoning. Eventually, we will reach some local state q''_i with $\delta_i(q''_i) = y'' \in \mathcal{V}$, because P is wait-free (i.e., there is no infinite execution ending with only inner i-actions). ◀

Putting all the previous lemmas together, we obtain Theorem 12:

▶ **Theorem 12.** *The semantics $[\![P]\!]$ of a wait-free protocol is a concurrent specification.*

This theorem ensures that the axioms of concurrent specifications are reasonable, in the sense that they are validated in our model. Thus, our concurrent specifications are the only ones of interest: specifications that do not satisfy these axioms cannot be implemented. For instance, this theorem implies that any protocol implementing a COUNT-like object *must* accept execution (c), since it is obtained by expanding a valid sequential execution, and the protocol's behavior is closed under expansion.

4 Linearizability

The notion of *linearizability* has been introduced by Herlihy and Wing [10] as a correctness criterion for concurrent implementations of sequential data structures. Linearizability is very popular thanks to its *locality* property, which allows programmers to reason modularly about each object. Here, we adopt a slightly unusual point of view on linearizability: instead of a correctness criterion, it is a map that turns a sequential specification into a concurrent specification, in our sense.

For ease of presentation, we begin by introducing a general notion called $\mathcal{L}$-linearizability which subsumes several variants found in the literature. It is actually quite straightforward to check that the usual notions of linearizability, set-linearizability and interval-linearizability are recovered by instantiating $\mathcal{L}$ with the right set of traces. Thus, the definition of $\mathcal{L}$-linearizability should not be regarded as a contribution of this paper; it is merely a presentation trick to avoid dealing with three variants of the same definition.

The main result of this section is stated in Theorem 16. We then explore its consequences in three particular cases: standard, set- and interval-linearizability.

4.1 $\mathcal{L}$-specifications

We first introduce a notion of specification, akin to the one of Definition 1, which is parameterized by a set $\mathcal{L}$ of traces, which we abstractly consider as the "linear" ones. To recover the standard notion of linearizability, we let $\mathcal{L}$ be the set of sequential traces, in which case $\mathcal{L}$-specifications correspond to sequential specifications.

We always require $\mathcal{L}$ to be alternating, prefix-closed, non-empty and

- *receptive* for complete traces: if $T \in \mathcal{L}$ is complete then $T \cdot \mathsf{i}_i^x \in \mathcal{L}$ for all $i \in [n]$ and $x \in \mathcal{V}$,
- *fully total*: if $T \in \mathcal{L}$ and $\pi_i(T)$ has a pending invocation then $T \cdot \mathsf{r}_i^x \in \mathcal{L}$ for every $x \in \mathcal{V}$.

Intuitively, these conditions mean that $\mathcal{L}$ is a set of traces which have some specific "shape" (e.g., being sequential), but it does not say anything about the values. The set of sequential traces is the smallest set satisfying those properties. Now, given such a set $\mathcal{L}$, an $\mathcal{L}$-specification says which execution traces are correct or not, but only among those of $\mathcal{L}$.

▶ **Definition 13** ($\mathcal{L}$-specification)**.** An *$\mathcal{L}$-specification* σ is a set of traces in $\mathcal{L}$ which is prefix-closed, non-empty and

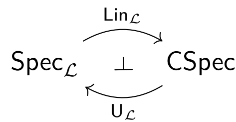
- *receptive* within $\mathcal{L}$: for all $T \in \sigma$, $i \in [n]$ and $x \in \mathcal{V}$, if $T \cdot \mathsf{i}_i^x \in \mathcal{L}$ then $T \cdot \mathsf{i}_i^x \in \sigma$,
- *total*: if $T \in \sigma$ and $\pi_i(T)$ has a pending invocation, there exists $x \in \mathcal{V}$ such that $T \cdot \mathsf{r}_i^x \in \sigma$.

We write $\mathsf{Spec}_\mathcal{L}$ for the set of $\mathcal{L}$-specifications.

In particular, a concurrent specification is an $\mathcal{L}$-specification for $\mathcal{L}$ the set of all alternating traces, such that conditions 6, 7 and 8 are moreover satisfied.

4.2 Comparison with concurrent specifications

In order to compare $\mathcal{L}$-specifications and concurrent ones, we define two functions as below:

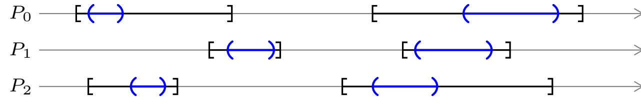


The function $\mathsf{U}_\mathcal{L}$ is defined by keeping only linear traces in a specification: $\mathsf{U}_\mathcal{L}(\tau) = \tau \cap \mathcal{L}$. The converse will require us to consider traces which are linearizable with respect to $\mathcal{L}$.

We write $T \rightsquigarrow T'$ when the trace T' can be obtained from the trace T by applying the following series of local transformations (forming a string rewriting system):

$$\mathsf{i}_i^x \cdot \mathsf{i}_j^y \rightsquigarrow \mathsf{i}_j^y \cdot \mathsf{i}_i^x \qquad \mathsf{r}_i^x \cdot \mathsf{r}_j^y \rightsquigarrow \mathsf{r}_j^y \cdot \mathsf{r}_i^x \qquad \mathsf{i}_i^x \cdot \mathsf{r}_j^y \rightsquigarrow \mathsf{r}_j^y \cdot \mathsf{i}_i^x \qquad \text{for } i \neq j.$$

This amounts to contracting the intervals, the opposite of expansion. In the picture below, T is represented in black with square brackets, T' in blue with round brackets:



▶ **Definition 14.** A trace $T \in \mathcal{T}$ is $\mathcal{L}$-*linearizable* w.r.t. an $\mathcal{L}$-specification σ if there exists a completion T' of T, obtained by appending responses to the pending invocations of T, and a trace $S \in \sigma$, such that $T' \rightsquigarrow S$.

It is not difficult to check that the above definition is equivalent to the usual one of [10]; this local presentation as a rewriting system is sometimes used in linearizability proofs using Lipton's left/right movers [14]. A minor difference is that the completion T' of T is usually allowed to remove some of the pending invocations of T. This is not useful here since all our specifications are total: a response to the pending invocations can always be found. Finally, given an $\mathcal{L}$-specification σ, we define $\mathsf{Lin}_{\mathcal{L}}(\sigma)$ as the set of traces which are $\mathcal{L}$-linearizable with respect to σ. We now show that it is a concurrent specification.

▶ **Proposition 15.** *Let* $\sigma \in \mathsf{Spec}_{\mathcal{L}}$ *be an* $\mathcal{L}$*-specification. Then*

$$\mathsf{Lin}_{\mathcal{L}}(\sigma) \quad = \quad \{T \in \mathcal{T} \mid T \text{ is linearizable with respect to } \sigma\}$$

is a concurrent specification.

Proof (Sketch). We prove the conditions 6, 7 and 8 of Definition 1, which are the ones of interest in this paper. The closure under prefix is a bit technical.

- Commutativity of invocations. Assume $T = U \cdot \mathsf{i}_i^x \cdot \mathsf{i}_j^y \cdot V \in \mathsf{Lin}_{\mathcal{L}}(\sigma)$. So there is a sequence of responses $\hat{T}$ and $S \in \sigma$ such that $T \cdot \hat{T} \rightsquigarrow S$. By rewriting one step ($i \neq j$ because of alternation), we get $U \cdot \mathsf{i}_j^y \cdot \mathsf{i}_i^x \cdot V \cdot \hat{T} \rightsquigarrow T \cdot \hat{T} \rightsquigarrow S$. Thus, $U \cdot \mathsf{i}_j^y \cdot \mathsf{i}_i^x \cdot V \in \mathsf{Lin}_{\mathcal{L}}(\sigma)$.
- Commutativity of responses is similar.
- For closure under expansion, assume $T = U \cdot \mathsf{r}_j^y \cdot \mathsf{i}_i^x \cdot V \in \mathsf{Lin}_{\mathcal{L}}(\sigma)$ with $i \neq j$, and let $\hat{T}$ and $S \in \sigma$ be such that $T \cdot \hat{T} \rightsquigarrow S$. We want to show that $T' = U \cdot \mathsf{i}_i^x \cdot \mathsf{r}_j^y \cdot V \in \mathsf{Lin}_{\mathcal{L}}(\sigma)$. But since $T' \cdot \hat{T} \rightsquigarrow T \cdot \hat{T} \rightsquigarrow S$, we are done. ◀

Proposition 15 shows that all the axioms that we impose on our concurrent specifications are reasonable in the sense that they are naturally enforced by all the specification techniques based on linearizability. Thus, linearizability is a canonical way of turning a weaker specification σ (e.g., one specifying only sequential behaviors) into a concurrent specification:

▶ **Theorem 16.** *The functions* $\mathsf{Lin}_{\mathcal{L}}$ *and* $\mathsf{U}_{\mathcal{L}}$ *are monotonous w.r.t. inclusions, and form a Galois connection: for every* $\sigma \in \mathsf{Spec}_{\mathcal{L}}$ *and* $\tau \in \mathsf{CSpec}$*,*

$$\mathsf{Lin}_{\mathcal{L}}(\sigma) \subseteq \tau \qquad \Longleftrightarrow \qquad \sigma \subseteq \mathsf{U}_{\mathcal{L}}(\tau).$$

Proof. The monotonicity of $\mathsf{U}_{\mathcal{L}}$ is trivial. For $\mathsf{Lin}_{\mathcal{L}}$, assume $\sigma \subseteq \sigma'$ and $T \in \mathsf{Lin}_{\mathcal{L}}(\sigma)$. Let $S \in \sigma$ be a linearization of T. Since $S \in \sigma'$, we also have $T \in \mathsf{Lin}_{\mathcal{L}}(\sigma')$.

Now let σ and τ as in the theorem and assume $\mathsf{Lin}_{\mathcal{L}}(\sigma) \subseteq \tau$. By monotonicity of $\mathsf{U}_{\mathcal{L}}$, $\mathsf{U}_{\mathcal{L}}(\mathsf{Lin}_{\mathcal{L}}(\sigma)) \subseteq \mathsf{U}_{\mathcal{L}}(\tau)$. But $\sigma \subseteq \mathsf{U}_{\mathcal{L}}(\mathsf{Lin}_{\mathcal{L}}(\sigma))$ since every $T \in \sigma$ is in $\mathcal{L}$ and is its own linearization (if T has pending invocations, we can add any valid response using totality).

Conversely, assume $\sigma \subseteq \mathsf{U}_{\mathcal{L}}(\tau)$, and let T be a linearizable trace w.r.t. σ. Let T' be an extension of T and $S \in \sigma \subseteq \mathsf{U}_{\mathcal{L}}(\tau) \subseteq \tau$ such that $T' \rightsquigarrow S$. So we can go from S to T' through a sequence of expansions and commutations, which gives us $T' \in \tau$ by applying axioms (6-8) of concurrent specifications, and by prefix closure, $T \in \tau$. ◀

In general, the posets $\mathsf{Spec}_{\mathcal{L}}$ and CSpec ordered by inclusion are not isomorphic, but the above theorem shows that the next best thing one could expect happens: every $\mathcal{L}$-specification has a canonical approximation as a concurrent specification and conversely. Note that Galois connections are widely used for comparing semantics of programs and deriving program analysis methods [3], and are a particular case of adjunctions, which have been promoted as the canonical way of comparing models for concurrency [18].

The right-to-left implication of Theorem 16 can be understood as follows: $\mathsf{Lin}_{\mathcal{L}}(\sigma)$ is the smallest concurrent specification which contains σ. Notice that the axioms (6-8) are crucial here: if one of them were missing, we could produce specifications smaller than or incomparable to $\mathsf{Lin}_{\mathcal{L}}(\sigma)$. In fact, the theorem states that $\mathsf{Lin}_{\mathcal{L}}$ is a kind of free construction: starting with the traces in σ, we add all the traces that are required to be in the specification by our axioms, and no other trace than the required ones.

4.3 Sequential linearizability

A trace T is *sequential* when the poset $(\mathrm{op}(T), \preceq)$ is totally ordered, we write seq for the set of sequential traces. A *sequential specification* is a seq-specification (i.e., Definition 13 with $\mathcal{L} = \mathsf{seq}$). Note that a sequential specification is not a particular case of concurrent specification: it only satisfies the receptivity condition of Definition 13, not the stronger one of Definition 1. Intuitively, this means a sequential specification does not specify which behaviors are allowed when some of the processes run in parallel. Sequential linearizability is thus a canonical way to extend a sequential specification to a concurrent one.

The notion of seq-linearizability coincides with the usual notion of linearizability, that we call here *sequential linearizability* to avoid confusion. The Galois connection of Theorem 16 says that $\mathsf{Lin}_{\mathsf{seq}}(\sigma)$ is the smallest concurrent specification whose set of sequential traces contains σ. Moreover, it is a Galois insertion:

▶ **Proposition 17.** *For every $\sigma \in \mathsf{Spec}_{\mathsf{seq}}$, we have $\mathsf{U}_{\mathsf{seq}}(\mathsf{Lin}_{\mathsf{seq}}(\sigma)) = \sigma$.*

Proof. The inequality $\sigma \subseteq \mathsf{U}_{\mathsf{seq}}(\mathsf{Lin}_{\mathsf{seq}}(\sigma))$ is implied by the Galois connection. Let $T \in \mathsf{U}_{\mathsf{seq}}(\mathsf{Lin}_{\mathsf{seq}}(\sigma))$, i.e., T is both sequential and linearizable w.r.t. σ. Let T' be a completion of T and $S \in \sigma$ such that $T' \rightsquigarrow S$. Since T' is sequential, it is a normal form of the rewriting system (i.e., no rule can be applied): so we must have $T' = S$. Moreover S is required to be in σ, so $T' \in \sigma$ and by prefix-closure, $T \in \sigma$. ◀

This implies that $\mathsf{Spec}_{\mathsf{seq}}$ is a subposet of CSpec, which justifies calling *linearizable* a concurrent specification in the image of $\mathsf{Lin}_{\mathsf{seq}}$. This is however a strict subposet, as an application of Theorem 16:

▶ **Proposition 18.** *There are non-linearizable concurrent specifications.*

Proof. From Proposition 17, any linearizable concurrent specification $\tau = \mathsf{Lin}_{\mathsf{seq}}(\sigma)$ satisfies $\mathsf{Lin}_{\mathsf{seq}}(\mathsf{U}_{\mathsf{seq}}(\tau)) = \tau$. Now, consider the concurrent specification SET-COUNT $\subseteq \mathcal{T}$ which is the set of traces whose set of operations has a partition $(E_i)_{i \in I}$ such that every $e, e' \in E_i$ are concurrent and their response value is the cardinal of E_i.

Informally, E_i is a set of pairwise-concurrent processes that "saw" each other. Note that, because of expansion, we cannot require that all processes running in parallel should see each other: the execution (b) is accepted. All other executions (a), (c), (d) and (e) are rejected. In a sequential trace $T \in$ SET-COUNT, every response returns 1. Thus, $\mathsf{Lin}_{\mathsf{seq}}(\mathsf{U}_{\mathsf{seq}}(\text{SET-COUNT}))$ only contains traces whose response is 1. But SET-COUNT also has traces with different responses, e.g. $\mathsf{i}_i \cdot \mathsf{i}_j \cdot \mathsf{r}_j^2 \cdot \mathsf{r}_i^2$. Therefore, SET-COUNT is not linearizable. ◀

4.4 Set-linearizability

The idea behind set-linearizability [17] is to specify what happens when a set of processes call an object at the same time. In this setting, an execution trace will be a sequence of sets of processes. In each of these sets, all the processes start executing, then all of them must terminate before we proceed with the next set of processes. Set-linearizability was also recently re-discovered by Hemed et al. [7], who call it *concurrency-aware linearizability*. A typical example of a set-linearizable (but not linearizable) object is the *immediate-snapshot* protocol [1] widely used in distributed computability [9]. Another example is Java's *exchanger* that allows two concurrent threads to atomically swap values.

A trace T is *set-sequential* if it is of the form $T = I_1 \cdot R_1 \cdots I_k \cdot R_k$, where each of the I_i is a non-empty sequence of invocations, R_i for $i < k$ is a sequence of responses with the same set of process numbers as I_i, and R_k is a (possibly empty) sequence of responses whose process numbers are included in those of I_k. We write set for the set of such traces. If we consider $\mathcal{L}$-linearizability with $\mathcal{L} = \mathsf{set}$, we recover the previously defined notion of set-linearizability [17]. Of course, Theorem 16 still holds, but we do not have an analogue of Proposition 17 here: given $\sigma \in \mathsf{Spec}_{\mathsf{set}}$, $\mathsf{U}_{\mathsf{set}}(\mathsf{Lin}_{\mathsf{set}}(\sigma))$ might actually contain more set-sequential traces than σ. This is because set-sequential specifications are not required to satisfy the expansion property nor the commutativity of invocations and of responses. The linearizability map adds just enough set-sequential traces to make these properties verified. A concurrent specification is *set-linearizable* if it is of the form $\mathsf{Lin}_{\mathsf{set}}(\sigma)$ for some $\sigma \in \mathsf{Spec}_{\mathsf{set}}$.

▶ **Example 19.** The SET-COUNT specification defined in the proof of Proposition 18 is set-linearizable. It is obtained as $\mathsf{Lin}_{\mathsf{set}}(\sigma)$ where σ is the set of set-sequential traces of the form $I_1 \cdot R_1 \cdots I_k \cdot R_k$, where every response in R_i is returning the value $|I_i|$.

▶ **Proposition 20.** *There are non-set-linearizable concurrent specifications.*

Proof. This is again an application of Theorem 16: from the properties of Galois connections, a set-linearizable concurrent specification τ must be such that $\mathsf{Lin}_{\mathsf{set}}(\mathsf{U}_{\mathsf{set}}(\tau)) = \tau$. Define the concurrent specification INTERVAL-COUNT as the set of alternating traces such that every operation e has a response of the form r_i^k, where k is smaller or equal to the number of operations that are overlapping with e. This is a very permissive version of counting: all executions (a)–(e) are allowed. In a set-sequential trace, all return values are at most n, the total number of processes. Therefore, $\mathsf{Lin}_{\mathsf{set}}(\mathsf{U}_{\mathsf{set}}(\tau))$ only contains traces whose response is smaller than n. But τ also contains traces with greater responses, as in execution (b) where process P_1 responds 3 even though only two processes are running. We deduce that INTERVAL-COUNT is not set-linearizable. ◀

4.5 Interval-linearizability

Interval-linearizability was introduced in [2] with the aim of going beyond linearizability and set-linearizability, in order to be able to specify every distributed *task*. They prove that every task can be obtained as the restriction to one-shot executions of an interval-linearizable object. We will show that this result actually extends beyond one-shot tasks: every concurrent specification is interval-linearizable. An example of an interval-linearizable (but not set-linearizable) object is the (non-immediate) write-snapshot object.

We write int for the set of all alternating traces. The notions of *interval-sequential specification* and *interval-linearizability* defined in [2] coincide with $\mathcal{L}$-specifications and $\mathcal{L}$-linearizability where $\mathcal{L} = \mathsf{int}$. Interval-sequential specifications are almost the same as

concurrent specifications, except that they do not require the commutativity of invocations, commutativity of responses and expansion property to be satisfied. In fact, it is mentioned in [2] that one can without loss of generality restrict to interval-sequential executions of the form $I_1 \cdot R_1 \cdots I_k \cdot R_k$, where the I_i (resp., R_i) are non-empty sets of invocations (resp., responses). This amounts to enforcing the commutativity of invocations and of responses. We do not include it here since it will be enforced anyway after we apply linearizability.

$\mathsf{U}_{\mathsf{int}}$ is by definition the "identity" function. As in the case of set-linearizability, an analogue of Proposition 17 does not hold, but we have the converse:

▶ **Proposition 21.** *For every* $\tau \in \mathsf{CSpec}$*,* $\mathsf{Lin}_{\mathsf{int}}(\mathsf{U}_{\mathsf{int}}(\tau)) = \tau$*.*

Proof. The inclusion $\mathsf{Lin}_{\mathsf{int}}(\mathsf{U}_{\mathsf{int}}(\tau)) \subseteq \tau$ follows from the Galois connection. For the other inclusion, recall that $\mathsf{U}_{\mathsf{int}}$ is the identity, so we want to prove $\tau \subseteq \mathsf{Lin}_{\mathsf{int}}(\tau)$. But this is trivial since every trace is its own linearization. ◀

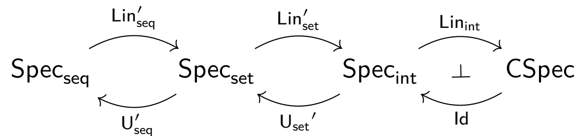
A concurrent specification is *interval-linearizable* if it is in the image of $\mathsf{Lin}_{\mathsf{int}}$.

▶ **Proposition 22.** *Every concurrent specification is interval-linearizable.*

Proof. This is an immediate corollary of Proposition 21: $\tau = \mathsf{Lin}_{\mathsf{int}}(\tau)$, and τ is (in particular) an interval-sequential specification. ◀

Thus, interval-linearizable objects and concurrent specifications are one and the same. One can see interval-linearizability as a convenient way of defining concurrent specifications in practice: we do not have to worry about the expansion and commutativity conditions (6-8), linearizability does that for us.

The results of the last three sections can be summed up in the following diagram, by factoring $\mathsf{Lin}_{\mathsf{seq}}$ and $\mathsf{Lin}_{\mathsf{set}}$ as follows:



where the ⊣ symbol between $\mathsf{Lin}_{\mathsf{int}}$ and Id indicates that they are related by a Galois connection, and the primed functions are the expected ones. Moreover, the two other Galois connections are obtained by composition:

$$\mathsf{Lin}_{\mathsf{int}} \circ \mathsf{Lin}'_{\mathsf{set}} = \mathsf{Lin}_{\mathsf{set}} \dashv \mathsf{U}_{\mathsf{set}} \qquad \text{and} \qquad \mathsf{Lin}_{\mathsf{int}} \circ \mathsf{Lin}'_{\mathsf{set}} \circ \mathsf{Lin}'_{\mathsf{seq}} = \mathsf{Lin}_{\mathsf{seq}} \dashv \mathsf{U}_{\mathsf{seq}}.$$

4.6 Other variants

Using the results of Section 4.2, it is easy to come up with new notions of linearizability. For instance, say a trace T is *k-concurrent* if every prefix of T has at most k pending invocations. Intuitively, a trace is k-concurrent if at any time, no more than k processes are running in parallel. Let $\mathcal{L}$ be the set of k-concurrent traces. Then, given a specification that says how an object behaves when it is accessed concurrently by at most k processes, $\mathcal{L}$-linearizability is a canonical way of extending this specification to any number of processes. In their paper about concurrency-aware linearizability [7], Hemed et al. specify Java's exchanger object on traces that are both 2-concurrent and set-sequential, then they apply linearizability to obtain the full concurrent specification.

5 Conclusion and perspectives

We have studied a class of concurrent specifications satisfying a number of desirable properties, and argued that those properties make sense computationally. This has allowed us to formally compare different notions of linearizability in Theorem 16. Finally, we have explored some consequences and applications of this theorem. We believe that this should be the starting point of a series of future works.

While we have illustrated the robustness of this notion, there are many possible small interesting variants, such as removing the totality condition to model programs which are not wait-free. In particular, it is often desirable to impose specifications to be deterministic: the most simple definition (if the specification contains two traces $T \cdot \mathsf{r}_i^x$ and $T \cdot \mathsf{r}_i^y$ then $x = y$ should hold) does not play well with asynchrony, we think that a definition in the spirit of [15] is however possible. Another direction worth considering is to get rid of the notion of global time, inherent to the trace formalism. Very recent work generalize linerizability in this direction, with applications to weak memory models [21] and relativistic effects [5].

We would also like to use our formalism in order to be able to reason about concurrent programs in a compositional way. The model we introduce in Section 3 is (purportedly) very close to the models of programming languages traditionally considered in game semantics [11], and in particular the asynchronous variants [15], which are able to express the main operational features of the languages while enjoying compositionality. We shall investigate the compositional (categorical) structures enjoyed by our concurrent specifications in the future.

References

1. Elizabeth Borowsky and Eli Gafni. Immediate Atomic Snapshots and Fast Renaming. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing*, PODC '93, pages 41–51. ACM, 1993.
2. Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. Specifying Concurrent Problems: Beyond Linearizability and up to Tasks. In *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, Proceedings*, pages 420–435, 2015.
3. Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252, 1977.
4. Ivana Filipović, Peter O'Hearn, Noam Rinetzky, and Hongseok Yang. Abstraction for concurrent objects. *Theoretical Computer Science*, 411(51):4379–4398, 2010. European Symposium on Programming 2009.
5. Seth Gilbert and Wojciech Golab. Making Sense of Relativistic Distributed Systems. In Fabian Kuhn, editor, *Distributed Computing, DISC 2015*, volume 8784 of *LNCS*, pages 361–375. Springer Berlin Heidelberg, 2014.
6. 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.
7. Nir Hemed, Noam Rinetzky, and Viktor Vafeiadis. Modular Verification of Concurrency-Aware Linearizability. In *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, Proceedings*, pages 371–387, 2015.
8. Maurice Herlihy. Wait-free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

9 Maurice Herlihy, Dmitry Kozlov, and Sergio Rajsbaum. *Distributed Computing Through Combinatorial Topology*. Morgan Kaufmann Publishers Inc., 2013.

10 Maurice Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

11 J Martin E Hyland and C-HL Ong. On full abstraction for PCF: I, II, and III. *Information and computation*, 163(2):285–408, 2000.

12 L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.

13 Leslie Lamport. On interprocess communication. *Distributed Computing*, 1(2):77–85, 1986.

14 Richard J Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.

15 Paul-André Melliès and Samuel Mimram. Asynchronous games: innocence without alternation. In *International Conference on Concurrency Theory*, pages 395–411. Springer, 2007.

16 J. Misra. Axioms for memory access in asynchronous hardware systems. In Stephen D. Brookes, Andrew William Roscoe, and Glynn Winskel, editors, *Seminar on Concurrency*, pages 96–110. Springer Berlin Heidelberg, 1985.

17 Gil Neiger. Set-Linearizability. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, page 396, 1994.

18 Mogens Nielsen. Models for concurrency. In *International Symposium on Mathematical Foundations of Computer Science*, pages 43–46. Springer, 1991.

19 Christos H. Papadimitriou. The Serializability of Concurrent Database Updates. *Journal of the ACM*, 26(4):631–653, 1979.

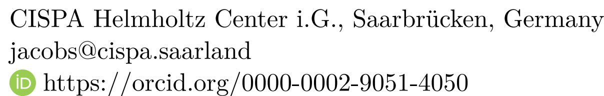
20 M. Raynal, G. Thia-Kime, and M. Ahamad. From serializable to causal transactions for collaborative applications. In *Proceedings of the 23rd EUROMICRO Conference*, pages 314–321, 1997.

21 G. Smith, K. Winter, and R. J. Colvin. A sound and complete definition of linearizability on weak memory models. *ArXiv e-prints*, February 2018. `arXiv:1802.04954`.

Parameterized Synthesis of Self-Stabilizing Protocols in Symmetric Rings

Nahal Mirzaie
University of Tehran, North Kargar St., Tehran, Iran
mirzaienahal@ut.ac.ir

Fathiyeh Faghih
University of Tehran, North Kargar St., Tehran, Iran
f.faghih@ut.ac.ir

Swen Jacobs[1]
CISPA Helmholtz Center i.G., Saarbrücken, Germany
jacobs@cispa.saarland

https://orcid.org/0000-0002-9051-4050

Borzoo Bonakdarpour[2]
Iowa State University, 207 Atanasoff Hall, Ames, IA 50011, USA
borzoo@iastate.edu

https://orcid.org/0000-0003-1800-5419

Abstract

Self-stabilization in distributed systems is a technique to guarantee *convergence* to a set of *legitimate states* without external intervention when a transient fault or bad initialization occurs. Recently, there has been a surge of efforts in designing techniques for automated synthesis of self-stabilizing algorithms that are correct by construction. Most of these techniques, however, are not parameterized, meaning that they can only synthesize a solution for a fixed and predetermined number of processes. In this paper, we report a breakthrough in parameterized synthesis of self-stabilizing algorithms in symmetric rings. First, we develop tight cutoffs that guarantee (1) closure in legitimate states, and (2) deadlock-freedom outside the legitimates states. We also develop a sufficient condition for convergence in *silent* self-stabilizing systems. Since some of our cutoffs grow with the size of local state space of processes, we also present an automated technique that significantly increases the scalability of synthesis in symmetric networks. Our technique is based on SMT-solving and incorporates a loop of synthesis and verification guided by counter-examples. We have fully implemented our technique and successfully synthesized solutions to maximal matching, three coloring, and maximal independent set problems.

2012 ACM Subject Classification Theory of computation → Logic and verification, Computer systems organization → Dependable and fault-tolerant systems and networks

Keywords and phrases Parameterized synthesis, Self-stabilization, Formal methods

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.29

[1] This research was supported by the German Research Foundation (DFG) under the project ASDPS (JA 2357/2-1).
[2] This research has been partially supported by the NSF SaTC-1813388 and a grant from Iowa State University.

© Nahal Mirzaie, Fathiyeh Faghih, Swen Jacobs, and Borzoo Bonakdarpour;
licensed under Creative Commons License CC-BY
22nd International Conference on Principles of Distributed Systems (OPODIS 2018).
Editors: Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira; Article No. 29; pp. 29:1–29:17

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

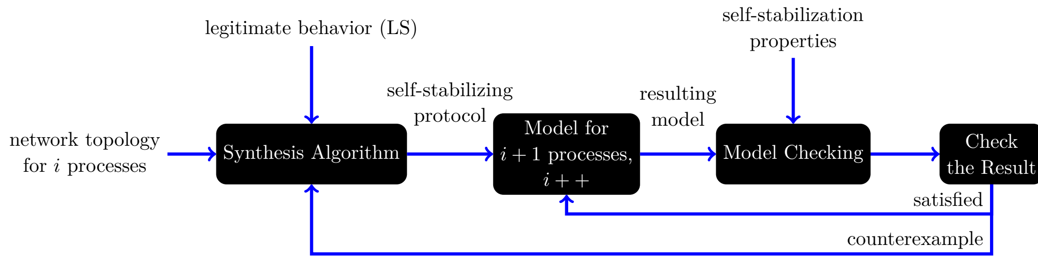


Figure 1 SMT-based Counterexample-Guided Synthesis Technique.

1 Introduction

Program *synthesis* (often called the *"holy grail"* of computer science) is the problem of automated generation of a computer program from a formally specified set of properties. The program generated in this fashion is guaranteed to be correct by construction. Program synthesis is known to be computationally intractable and, thus, is usually used to deal with small but intricate components of a system, e.g., concurrent/distributed algorithms that may exhibit obscure corner cases, where reasoning about their correctness is not straightforward.

Dijkstra [3] introduced the notion of *self-stabilization* in distributed systems, where the system always converges to a good behavior even if it is arbitrarily initialized or is subject to transient faults. Proof of self-stabilization is, however, often much more complex than what it initially seems like. Dijkstra himself published the proof of correctness of his seminal 3-state machine solution 12 years later [4]. This means that program synthesis can play a prime role in designing and reasoning about the correctness of self-stabilizing algorithms. In [8, 9, 11, 12, 10], we introduced a set of algorithms and tools for synthesizing self-stabilizing protocols. Our techniques take as input the network topology, timing model (asynchronous or synchronous), the good behavior of the protocol (either explicitly as a set of *legitimate states* or implicitly as a set of temporal logic formulas), type of symmetry, and type of stabilization (e.g., strong, weak, monotonic, ideal) and generate a set of first-order modulo theory constraints. Then, a satisfiability modulo theory (SMT) solver solves these constraints and, if satisfiable, produces a model that respects the input specification. Our tool Assess [10] has successfully synthesized complex algorithms such as Raymond's distributed mutual exclusion [24], Dijkstra's token ring [3] (for both three and four state machines), maximal matching [23], weak stabilizing token circulation in anonymous networks [2], and the three coloring problem [14]. Our algorithms are *complete* for a predetermined fixed number of processes; i.e., if they fail to find a solution to the synthesis problem, then there does not exist one. This completeness, however, comes at a big cost which is scalability. That is, for most instances, we could only synthesize solutions up to 5 processes at best.

In this paper, our goal is to address scalability as well as the shortcoming that the previous work can synthesize only a fixed and predetermined number of processes. To this end, we focus on automated synthesis of self-stabilizing protocols in *symmetric* and *parameterized* rings, where an unbounded number of processes exhibit identical behavior. We make two main contributions. First, we show how to solve the *parameterized synthesis problem* based on the notion of *cutoffs* [6] that can guarantee properties of distributed systems of arbitrary size by considering only systems of up to a certain fixed size $c \in \mathbb{N}$, and augment this by a sound but incomplete abstraction-based synthesis approach for properties that are known to be undecidable. In particular, we provide:

- cutoffs for the closure and deadlock-freedom properties, under the assumption that the set of legitimate states is defined by a conjunction of predicates on the local state of processes; we show that smaller cutoffs are possible under additional assumptions, and that all our cutoffs are tight under the assumptions we consider.
- an abstraction-based method for the convergence property, which is known to be undecidable in general [19, 21]; we show how, for the class of silent algorithms, a sufficient condition for convergence of the parameterized system can be efficiently checked on a finite system that over-approximates the behavior of systems of arbitrary size.

Note that these results can be used for both synthesis and verification. A drawback of our cutoffs is that they are quadratic in the state space of a single process, so even with a cutoff, we need synthesis methods that scale to a large number of processes. Thus, as our second contribution, we propose a counterexample-guided synthesis technique that exploits our symmetry assumption. More specifically, it consists of fours steps (see Fig. 1):

1. First, we *synthesize* a solution for a small network of i processes using existing techniques;
2. Next, we trivially generalize this solution to a larger network with $i+1$ processes;
3. Then, we *verify* this solution using a model checker, and
4. If verification succeeds, we return to step 2 to attempt a larger network. Otherwise, we obtain a counterexample that is added as a negative constraint for the synthesis algorithm, and we return to step 1 for another round of synthesis with limited search space.

Using this approach and our cutoff results, we successfully synthesized parameterized self-stabilizing protocols for well-known problems including three coloring, maximal matching, and maximal independent set in less than 10 minutes. To our knowledge, this is the first instance of such parameterized synthesis.

Organization. The rest of the paper is organized as follows. Section 2 introduces the preliminary concepts. In Section 3, we present the formal statement of our synthesis problem. The parameterized correctness results are presented in Section 4, while our counterexample-guided synthesis approach is presented in Section 5. Experimental results and case studies are reported in Section 6. Related work is discussed in Section 7, and finally, we make concluding remarks and discuss future work in Section 8.

2 Preliminaries

2.1 Distributed Programs

Most self-stabilizing algorithms are defined in the shared-memory model. Assume V to be the set of all variables in the system, where each variable $v \in V$ has a finite domain D_v. We define a *state* s as a valuation of each variable in V by a value in its domain. The set of all possible states is called the *state space*, and represented by S. A *transition* is defined as an ordered pair (s_0, s_1), where $s_0, s_1 \in S$. We denote the value of a variable v in state s by $v(s)$.

▶ **Definition 1.** A *process* π is a tuple $\langle R_\pi, W_\pi, T_\pi \rangle$, where

- $R_\pi \subseteq V$ is the set of variables that π can read their values and is called the *read-set* of π;
- $W_\pi \subseteq R_\pi$ is the set of variables that π can write to, and is called the *write-set* of π, and
- T_π is the set of transitions of π, where for each transition $(s_0, s_1) \in T_\pi$ and each variable v, such that $v(s_0) \neq v(s_1)$, we have $v \in W_\pi$.

The third condition imposes the constraint that a process can only change the value of a variable in its write-set, and the second condition states that this change cannot be blind. A process $\pi = \langle R_\pi, W_\pi, T_\pi \rangle$ is called *enabled* in state s_0, if there exists a state s_1, such that $(s_0, s_1) \in T_\pi$. The *local state space* of π is the set of all possible valuations of the variables that π can read: $S_\pi = \prod_{v \in R_\pi} D_v$

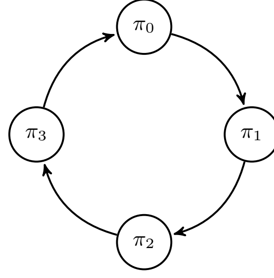


Figure 2 One-bit maximal matching example.

▶ **Definition 2.** A *distributed program* is a tuple $\mathcal{D} = \langle P_\mathcal{D}, T_\mathcal{D} \rangle$, where

- $P_\mathcal{D}$ is a set of processes over a common set V of variables, such that:
  - for any two distinct processes $\pi_1, \pi_2 \in P_\mathcal{D}$, we have $W_{\pi_1} \cap W_{\pi_2} = \emptyset$;
  - for each process $\pi \in P_\mathcal{D}$ and each transition $(s_0, s_1) \in T_\pi$, the following *read restriction* holds:

$$\forall s_0', s_1' : ((\forall v \in R_\pi : (v(s_0) = v(s_0') \ \wedge \ v(s_1) = v(s_1'))) \wedge (\forall v \notin R_\pi : v(s_0') = v(s_1'))) \implies (s_0', s_1') \in T_\pi \quad (1)$$

- $T_\mathcal{D}$ is the set of transitions and is the union of transitions of all processes:

$$T_\mathcal{D} = \bigcup_{\pi \in P_\mathcal{D}} T_\pi$$

Intuitively, the read restriction in Definition 2 imposes the constraint that for each process π, each transition in T_π depends only on the variables in the read-set of π. Thus, each transition is an equivalence class in $T_\mathcal{D}$, which we call a *group* of transitions. The key consequence of read restrictions is that during synthesis, if a transition is included (respectively, excluded) in $T_\mathcal{D}$, then its corresponding group must also be included (respectively, excluded) in $T_\mathcal{D}$.

Example. We use the problem of distributed self-stabilizing *one-bit maximal matching* as a running example to describe the concepts throughout the paper. Consider a ring of 4 processes (see Fig. 2), and let $V = \{x_0, x_1, x_2, x_3\}$ be the set of variables, where each x_i is a Boolean variable with domain $\{\mathtt{F}, \mathtt{T}\}$. Let $\mathcal{D} = \langle P_\mathcal{D}, T_\mathcal{D} \rangle$ be a distributed program, where $P_\mathcal{D} = \{\pi_0, \pi_1, \pi_2, \pi_3\}$. Each process π_i $(0 \le i \le 3)$ can write to the variable x_i (i.e., $W_{\pi_i} = \{x_i\}$), and read the variables of its own and its neighbors ($R_{\pi_i} = \{x_i, x_{(i+1) \mod 4}, x_{(i-1) \mod 4}\}$). Notice that following Definition 2 and read/write restrictions of π_0, (arbitrary) transitions

$$t_1 = ([x_0 = \mathtt{F}, x_1 = \mathtt{F}, x_2 = \mathtt{F}, x_3 = \mathtt{F}], [x_0 = \mathtt{T}, x_1 = \mathtt{F}, x_2 = \mathtt{F}, x_3 = \mathtt{F}])$$
$$t_2 = ([x_0 = \mathtt{F}, x_1 = \mathtt{F}, x_2 = \mathtt{T}, x_3 = \mathtt{F}], [x_0 = \mathtt{T}, x_1 = \mathtt{F}, x_2 = \mathtt{T}, x_3 = \mathtt{F}])$$

are in the same group. The reason is that π_0 cannot read x_2, and if, for example, t_1 is included in the set of transitions, while t_2 is not, it implies that the execution in process π_0 depends on the value of x_2, which is not possible.

▶ **Definition 3.** An *uninterpreted local function* for a process maps the *local state space* of a process to a domain D_{lf}. The interpretation of an uninterpreted local function for a process π is a function:

$$S_\pi \to D_{lf}$$

where S_π is the local state space of π.

In the sequel, we use "uninterpreted functions" to refer to uninterpreted local functions.

Example. To formulate the requirements in the one-bit maximal matching example, we assume each process π_i is associated with an uninterpreted local function, called *match*, with the domain $D_{match_i} = \{l, r, n\}$, where l, r, and n correspond to the cases where the process is matched to its left, right, and no neighbor (self-matched), respectively. The interpretation of $match_i$ is a function:

$$(match_i)_{\mathrm{I}} : \{\mathtt{F}, \mathtt{T}\} \times \{\mathtt{F}, \mathtt{T}\} \times \{\mathtt{F}, \mathtt{T}\} \rightarrow \{l, r, n\}$$

In other words, the value of $match_i$ depends on the value of the process and its neighbors' Boolean variables.

▶ **Definition 4.** A *local predicate* of a process maps the *local state space* of a process to a Boolean:

$$S_\pi \rightarrow \{\mathtt{F}, \mathtt{T}\}$$

We use this definition to define legitimate states in Section 2.3.

2.1.1 Network Topology

A topology specifies the communication model of a distributed program.

▶ **Definition 5.** A *topology* is a tuple $\mathcal{T} = \langle V, |P_\mathcal{T}|, R_\mathcal{T}, W_\mathcal{T} \rangle$, where
- V is a finite set of finite-domain discrete variables,
- $|P_\mathcal{T}| \in \mathbb{N}_{\geq 1}$ is the number of processes,
- $R_\mathcal{T}$ is a mapping $\{0 \ldots |P_\mathcal{T}| - 1\} \rightarrow 2^V$ from a process index to its read-set,
- $W_\mathcal{T}$ is a mapping $\{0 \ldots |P_\mathcal{T}| - 1\} \rightarrow 2^V$ from a process index to its write-set, such that $W_\mathcal{T}(i) \subseteq R_\mathcal{T}(i)$, for all i $(0 \leq i \leq |P_\mathcal{T}| - 1)$.

Example. The topology of our maximal matching problem is a tuple $\langle V, |P_\mathcal{T}|, R_\mathcal{T}, W_\mathcal{T} \rangle$:
- $V = \{x_0, x_1, x_2, x_3\}$, with domains $D_{x_0} = D_{x_1} = D_{x_2} = D_{x_3} = \{\mathtt{T}, \mathtt{F}\}$,
- $|P_\mathcal{T}| = 4$,
- $R_\mathcal{T}(0) = \{x_0, x_1, x_3\}$, $R_\mathcal{T}(1) = \{x_1, x_2, x_0\}$,
  $R_\mathcal{T}(2) = \{x_2, x_3, x_1\}$, $R_\mathcal{T}(3) = \{x_3, x_0, x_2\}$, and
- $W_\mathcal{T}(0) = \{x_0\}$, $W_\mathcal{T}(1) = \{x_1\}$, $W_\mathcal{T}(2) = \{x_2\}$, and $W_\mathcal{T}(3) = \{x_3\}$.

▶ **Definition 6.** A distributed program $\mathcal{D} = \langle P_\mathcal{D}, T_\mathcal{D} \rangle$ *has* topology $\mathcal{T} = \langle V, |P_\mathcal{T}|, R_\mathcal{T}, W_\mathcal{T} \rangle$ iff
- each process $\pi \in P_\mathcal{D}$ is defined over V
- $|P_\mathcal{D}| = |P_\mathcal{T}|$
- there is a mapping $g : \{0 \ldots |P_\mathcal{T}| - 1\} \rightarrow P_\mathcal{D}$ such that

$$\forall i \in \{0 \ldots |P_\mathcal{T}| - 1\} : (R_\mathcal{T}(i) = R_{g(i)}) \wedge (W_\mathcal{T}(i) = W_{g(i)})$$

2.2 Symmetric Networks

Roughly speaking, a topology is symmetric, if the read-set and write-set of any two distinct processes can be swapped (i.e., there is a bijection that maps read/write variables of a process to another).

▶ **Definition 7.** A topology $\mathcal{T} = \langle V, |P_\mathcal{T}|, R_\mathcal{T}, W_\mathcal{T} \rangle$ is *symmetric*, iff for any distinct $i, j \in \{0 \ldots |P_\mathcal{T}| - 1\}$, there exists

- a bijection $f : R_{\mathcal{T}}(i) \to R_{\mathcal{T}}(j)$, such that $\forall v \in R_{\mathcal{T}}(i) :\ D_v = D_{f(v)}$, and
- a bijection $g : W_{\mathcal{T}}(i) \to W_{\mathcal{T}}(j)$, such that $\forall v \in W_{\mathcal{T}}(i) :\ D_v = D_{g(v)}$.

We call a symmetric topology a (bi-directional) *ring* (of size $k = |P_{\mathcal{T}}|$) if for every $i \in \{0 \dots |P_{\mathcal{T}}| - 1\}$, we have $R_{\mathcal{T}}(i) = W_{\mathcal{T}}(i-1 \mod k) \cup W_{\mathcal{T}}(i) \cup W_{\mathcal{T}}(i+1 \mod k)$. Since in this paper we only deal with rings, to simplify notation throughout the paper, arithmetic on process indices is implicitly modulo the size of the ring.

Example. The topology of our one-bit maximal matching example is symmetric, and a ring of size 4 (Fig. 2). For any two numbers i and j, function g is the mapping from x_i to a x_j, and function f maps $x_i \mapsto x_j$, $x_{(i+1)} \mapsto x_{(j+1)}$, and $x_{(i-1)} \mapsto x_{(j-1)}$.

▶ **Definition 8.** A distributed program $\mathcal{D} = \langle P_{\mathcal{D}}, T_{\mathcal{D}} \rangle$ is called *symmetric* iff

- it has a symmetric topology, and
- for any two distinct processes $\pi, \pi' \in P_{\mathcal{D}}$, the following condition holds:

$$\begin{aligned} &\forall (s_0, s_1) \in T_\pi :\ \exists (s'_0, s'_1) \in T_{\pi'} : \\ &\Big(\forall v \in R_\pi : \big(v(s_0) = f(v)(s'_0)\big)\big) \ \wedge\ \big(\forall v \in W_\pi : (v(s_1) = g(v)(s'_1)\big) \Big) \end{aligned} \tag{2}$$

where f and g are the functions defined in Definition 7.

In other words, in a symmetric distributed program the read- and write-sets of all processes are identical up to renaming, and so are their transitions. Therefore, we also write $\mathcal{T}^\pi$ for a symmetric distributed program that has topology $\mathcal{T}$ and where all processes are identical up to renaming to π.

2.3 Self-Stabilization

Given a subset of the state space, called the set of *legitimate states* (denoted by *LS*), a *self-stabilizing* [3] program always recovers to a state in *LS* from any arbitrary state (e.g., due to bad initialization or occurrence of transient faults) in a finite number of steps, and stays in *LS* thereafter.

▶ **Definition 9.** A *computation* of $\mathcal{D} = \langle P_{\mathcal{D}}, T_{\mathcal{D}} \rangle$ is an infinite sequence of states $\overline{s} = s_0 s_1 \cdots$, such that: (1) for all $i \geq 0$, we have $(s_i, s_{i+1}) \in T_{\mathcal{D}}$, and (2) if a computation reaches a state s_i, from where there is no state $\mathfrak{s} \neq s_i$, such that $(s_i, \mathfrak{s}) \in T_{\mathcal{D}}$, then the computation stutters at s_i indefinitely. Such a computation is called a *terminating computation.*

▶ **Definition 10.** A distributed program $\mathcal{D} = \langle P_{\mathcal{D}}, T_{\mathcal{D}} \rangle$ is *self-stabilizing* for a set *LS* of legitimate states iff

1. *(Convergence)* For any computation $\overline{s} = s_0 s_1 \cdots$, there exists a state $s_j \in \overline{s}$ $(j \geq 0)$, such that $s_j \in LS$.
2. *(Closure)* For any transition $(s_0, s_1) \in T_{\mathcal{D}}$, if $s_0 \in LS$, then $s_1 \in LS$.

▶ **Definition 11.** A distributed program $\mathcal{D} = \langle P_{\mathcal{D}}, T_{\mathcal{D}} \rangle$ is *silent* with respect to a given *LS* if for any transition $(s_0, s_1) \in T_{\mathcal{D}}$, if $s_0 \in LS$, then $s_1 = s_0$.

▶ **Definition 12.** A set of legitimate states is *locally defined* if it can be defined by

$$s \in LS \quad \text{if and only if} \quad \forall i \in \{0 \dots |P_{\mathcal{T}}| - 1\} : LS_i(s),$$

where LS_i is a predicate on the read-set of process π_i.

Example. In a directed graph, a maximal matching is a maximal set of edges, in which no two edges share a common vertex. In a ring topology, each process can be matched to one of its two adjacent processes. To formulate this requirement, we assume each process π_i is associated with a local uninterpreted function, called $match_i$, with the domain $D_{match_i} = \{l, r, n\}$. LS can be locally defined with

$$
\begin{aligned}
LS_i = \big\{s \mid\ & (match_{i-1}(\Pi_{R_{\pi_{i-1}}}(s)) = r \wedge match_i(\Pi_{R_{\pi_i}}(s)) = l \wedge match_{i+1}(\Pi_{R_{\pi_{i+1}}}(s)) = n) \vee \\
& (match_{i-1}(\Pi_{R_{\pi_{i-1}}}(s)) = n \wedge match_i(\Pi_{R_{\pi_i}}(s)) = r \wedge match_{i+1}(\Pi_{R_{\pi_{i+1}}}(s)) = l) \vee \\
& (match_{i-1}(\Pi_{R_{\pi_{i-1}}}(s)) = l \wedge match_i(\Pi_{R_{\pi_i}}(s)) = n \wedge match_{i+1}(\Pi_{R_{\pi_{i+1}}}(s)) = r) \vee \\
& (match_{i-1}(\Pi_{R_{\pi_{i-1}}}(s)) = l \wedge match_i(\Pi_{R_{\pi_i}}(s)) = r \wedge match_{i+1}(\Pi_{R_{\pi_{i+1}}}(s)) = l) \vee \\
& (match_{i-1}(\Pi_{R_{\pi_{i-1}}}(s)) = r \wedge match_i(\Pi_{R_{\pi_i}}(s)) = l \wedge match_{i+1}(\Pi_{R_{\pi_{i+1}}}(s)) = r)\big\}
\end{aligned}
$$

The system is in its legitimate state, if and only if all processes are in their local legitimate states. For example, in a ring of size three with the set of processes $P = \{\pi_0, \pi_1, \pi_2\}$, the set of legitimate states can be formulated as the following:

$$\{s \mid LS_0(s) \wedge LS_1(s) \wedge LS_2(s)\}$$

Note how uninterpreted functions can be used to easily express LS. Without $match_i$, the user has to explicitly specify the cases where a process is matched to its left, right or itself, using the Boolean variables of its own and its adjacent processes (its read set).

3 Problem Statement

Our goal is to propose an automated method for parameterized synthesis of silent self-stabilizing protocols in symmetric ring networks. That is, we consider a problem where the size of the topology is a parameter, and we want to automatically synthesize the transition predicate and the interpretation of the uninterpreted function of each process, such that the resulting distributed program is silent self-stabilizing for any value of the parameter.

Formally, a *parameterized topology* is a sequence of symmetric topologies $\mathcal{T}_1, \mathcal{T}_2, \ldots$, where for all n we have $|P_{\mathcal{T}_n}| = n$ and bijections read-sets and write-sets, as required in Definition 7, also exist between process indices from different elements of the sequence. A *parameterized program* is a sequence of symmetric distributed programs $\mathcal{D}_1, \mathcal{D}_2, \ldots$ such that $\mathcal{D}_i = \mathcal{T}_i^\pi$ for a parameterized topology $\mathcal{T}_1, \mathcal{T}_2, \ldots$, and some process π.

The *parameterized synthesis problem* takes as input:

- a parameterized topology, and
- a set of locally defined legitimate states LS,

and generates as output:

- a process π such that for every element $\mathcal{T}_n$ of the topology, the program $\mathcal{D}_n = \mathcal{T}_n^\pi$ is self-stabilizing to LS.

▶ **Definition 13.** For a given parameterized topology and a property under consideration, a *cutoff* is a natural number c such that for any given process π and a locally defined LS the following holds: $\mathcal{D}_n = \mathcal{T}_n^\pi$ satisfies the property wrt. LS for all $n \in \mathbb{N}$ iff $\mathcal{D}_i = \mathcal{T}_i^\pi$ satisfies the property wrt. LS for all $i \in \{1 \ldots c\}$.

Note that cutoffs can be used for both parameterized verification and synthesis. In Section 4, we will present cutoffs for two properties: i) closure, and ii) the absence of deadlocks outside of LS. Moreover, we will introduce an abstraction-based method that can be combined with the cutoffs to solve the parameterized synthesis problem.

4 Parameterized Synthesis of Self-Stabilization in Symmetric Rings

In this section, we show how to reduce reasoning about parameterized programs to reasoning about a finite number of finite programs. To prove self-stabilization, we need to prove that the algorithm has the two properties of closure and convergence. We split the latter into two properties: (1) the absence of deadlocks outside of LS, and (2) the absence of cycles outside of LS. In the following, we provide tight cutoffs for closure and deadlocks outside of LS, as well as a sound abstraction to prove the absence of cycles outside of LS. Finally, we provide our main theorem that combines these results into a method for parameterized synthesis of self-stabilizing algorithms in rings.

4.1 Cutoffs for Closure and Deadlock Detection

Assume that the write-set of each process has l valuations. In other words, if process π_i has $W_{\mathcal{T}}(i) = \{v_1, \ldots, v_n\}$, then $l = |D_{v_1}| \times \cdots \times |D_{v_n}|$.

▶ **Lemma 1.** For self-stabilizing algorithms on a ring topology, the following are cutoffs for the closure property:

- $c = l^2 + 1$, if LS is locally defined;
- $c = l + 1$, if LS is locally defined and LS_i only depends on $W_{\mathcal{T}}(i)$ and $W_{\mathcal{T}}(i+1)$, and
- $c = 3$, if LS is locally defined and LS_i only depends on $W_{\mathcal{T}}(i)$.

All of the cutoffs are tight under their respective assumptions.

▶ **Lemma 2.** For self-stabilizing algorithms on a ring topology, the following are cutoffs for the detection of deadlocks outside of LS:

- $c = l^2 + 1$, if LS is locally defined;
- $c = l + 1$, if LS is locally defined and transitions of processes only depend on $W_{\mathcal{T}}(i)$ and $W_{\mathcal{T}}(i+1)$ (i.e., the ring is uni-directional), and
- $c = 3$, if LS is locally defined and transitions of processes only depend on $W_{\mathcal{T}}(i)$ (i.e., processes are completely independent).

All of the cutoffs are tight under their respective assumptions.[3]

4.2 Process Abstraction for Convergence

As mentioned before, to prove self-stabilization of a parameterized program, we need to prove closure and convergence. Closure can be proved based on Lemma 1, and Lemma 2 shows how to deal with deadlocks outside of LS. Thus, the missing part is a method to prove that there are no cycles outside of LS that prevents a computation to eventually reach LS. In contrast to the two previous problems, we now consider infinite behaviors of the system. Since parameterized verification and synthesis of symmetric self-stabilization in rings is known to be undecidable [19, 21], we cannot obtain cutoffs for this property. Therefore, we resort to proving the absence of cycles based on a sound abstraction of the system behavior.

The basic idea is the following: we check whether there is a loop that starts and ends in the same *local* state for an arbitrary process. If we can show that this is not possible, then certainly no global loop is possible. Note that this is a stronger property than what we want to prove; it proves that there could not be any loops in the protocol, neither inside nor

[3] Detailed proofs for Lemmas 1 and 2 can be found at `http://web.cs.iastate.edu/~borzoo/Publications/18/OPODIS/opodis18.pdf`.

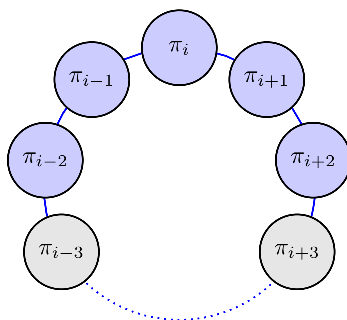


Figure 3 Blue processes act based on the algorithm and grey (with slanted lines) processes act randomly.

outside LS. It is obvious that this property can only be satisfied for silent protocols. To this end, we fix five processes (see Fig. 3), and define the following property:

$$S \Rightarrow \ (\Diamond\Box S \ \vee \ \neg\Box\Diamond S),$$

where S is the local state of π_i (i.e., the valuation of its read-set), $\Diamond$ is the 'eventually' operator and $\Box$ is the 'always' operator in temporal logic. That is, given a local state in S, any future extension either reaches a state where S is continually true, or, S does not become true infinitely often. Next, we attempt to prove the property in a ring of size 7, where 5 processes behave according to the synthesized protocol, and the other two processes have the same write-set, but can execute arbitrary transitions. The idea is that these two processes over-approximate the possible behavior of all other processes. If we can prove the property above in this abstraction of the system, then this implies that no loops are possible in a concrete system in a ring of size ≥ 5. Note that the precision of the abstraction can be refined by increasing the number of processes that behave according to the protocol, or by including the local state of additional processes into S. For the problems we considered in our experiments (see Sect. 6), the fixed abstraction with $5 + 2$ processes was sufficient.

4.3 Parameterized Self-Stabilization

Based on Lemmas 1 and 2, and the approach in Section 4.2, we obtain our main result.

▶ **Theorem 14.** *Let $\mathcal{T}_1, \mathcal{T}_2, \ldots$ be a parameterized ring topology, π a process, and let LS be locally defined by LS_i. Let c_1 and c_2 be cutoffs for closure and deadlock detection wrt. LS, respectively. If (1) closure holds in rings of size up to c_1, (2) deadlocks outside of LS are impossible in rings of size up to c_2, and (3) the absence of cycles can be proven in rings of up to size 4 and in an abstract system as above, then every instance of the parameterized program $\mathcal{D}_1 = \mathcal{T}_1^\pi, \mathcal{D}_2 = \mathcal{T}_2^\pi, \ldots$ is self-stabilizing to LS.*

5 SMT-based Counterexample-Guided Synthesis

5.1 General Idea

In [8, 9, 11], we introduced SMT-based methods to solve the synthesis problem for self-stabilizing systems. In a nutshell, our techniques generate a set of SMT constraints from the input synthesis instance and produce a model that represents a self-stabilizing protocol. In

order to scale up these technique to synthesize solutions up to the cutoff point efficiently, in this section, we propose a method, where we find a solution for a larger topology using a solution for a smaller topology. Let us first entertain a naïve idea, where we first synthesize a protocol for a small topology and then simply use this solution for larger topologies with the hope that since the protocol is symmetric, a small solution works in a larger network as well. We now show that this approach is not conceivable even for very simple protocols.

Example. When applying our latest algorithm [11] to the one-bit maximal matching example, the first synthesized solution for 4 processes is the following transition relation encoded by guarded commands for each process π_i:

$$\begin{array}{lrcl}\pi_i: & (x_i = \mathtt{F}) \wedge (x_{(i+1)} = \mathtt{F}) \wedge (x_{(i-1)} = \mathtt{F}) & \rightarrow & x_i := \mathtt{T} \\ & (x_i = \mathtt{T}) \wedge (x_{(i+1)} = \mathtt{T}) & \rightarrow & x_i := \mathtt{F}\end{array}$$

and the following interpretation for uninterpreted function $match_i$:

$$\begin{array}{lrcl}match_i: & (x_i = \mathtt{T}) \wedge \big((x_{(i+1)} = \mathtt{T}) \vee (x_{(i-1)} = \mathtt{F})\big) & \mapsto & l \\ & (x_{(i+1)} = \mathtt{T}) \wedge (x_{(i-1)} = \mathtt{F}) & \mapsto & l \\ & (x_i = \mathtt{T}) \wedge (x_{(i+1)} = \mathtt{F}) \wedge (x_{(i-1)} = \mathtt{T}) & \mapsto & r \\ & (x_i = \mathtt{F}) \wedge (x_{(i+1)} = \mathtt{T}) \wedge (x_{(i-1)} = \mathtt{T}) & \mapsto & r \\ & (x_i = \mathtt{F}) \wedge (x_{(i+1)} = \mathtt{F}) & \mapsto & n\end{array}$$

Now, if we trivially use the synthesized protocol on a topology with 5 processes, the resulting protocol is incorrect. In particular, the following is a counterexample (i.e., a finite computation that violates the specification) in terms of predicate *match*:

$$\Big([match_0 = n, match_1 = n, match_2 = n, match_3 = n, match_4 = n],$$
$$[match_0 = l, match_1 = n, match_2 = n, match_3 = n, match_4 = l],$$
$$[match_0 = l, match_1 = r, match_2 = l, match_3 = n, match_4 = l]\Big)$$

This computation violates convergence, as it reaches a deadlock state in $\neg LS$. This example shows that a synthesized symmetric solution cannot be trivially extended to larger topologies.

5.2 The Counterexample-Guided Synthesis Algorithm

In order to limit the search space of SMT-solvers for a solution, we incorporate a synthesis-verification loop guided by counterexamples. Our approach consists of the following steps:

1. Given a topology with i processes and a set of legitimate states, we use our existing approach [8, 9, 11] to formulate the synthesis problem as an SMT instance.
2. We use an SMT solver to find a solution for the SMT instance, as a transition relation and an interpretation for each uninterpreted function. Note that due to symmetry, the transition relations and the interpretation functions are identical for all processes.
3. Next, we generalize the solution for a topology with $i+1$ processes and verify this solution using a model checker.
4. If the result of verification is positive, we go back to step 3 to check the properties for a topology with $i+2$ processes. Otherwise, we transform the generated counterexample into an SMT constraint and add it to the initial SMT instance and return to step 2.

For reasons of space, we do not include the details of our SMT-based synthesis technique [8, 9, 11]. We now analyze the nature of counterexamples. In the context of closure and convergence, a model checker may generate a counterexample of the form $\overline{s} = s_0 s_1 \cdots s_n$. Observe that $\overline{s}$ is one of the following three types of counterexamples:

- If closure is violated, then $\overline{s} = (s_0, s_1)$, where $s_0 \in LS$ and $s_1 \notin LS$.
- If convergence is violated, $\overline{s} = s_0 s_1 \cdots s_n$, where for all $i \in [0, n]$, $s_i \notin LS$ and either
  - $s_0 = s_n$; i.e., a loop exists outside the set of legitimate states, or
  - there does not exist a state $\mathfrak{s}$, where $(s_n, \mathfrak{s})$ is a valid transition; i.e., s_n is a *deadlock* state outside the set of legitimate states.

Dealing with the first type of counterexamples is pretty straightforward: we only add a constraint to the SMT instance that disallows transition (s_0, s_1) in the transition relation. To address deadlocks, we need to add a constraint to the SMT instance to enforce a change in the resulting synthesized model, so that s_n is not a deadlock state. To this end, we propose two sets of heuristics to change either the transition relation or the interpretation of uninterpreted functions in Section 5.3. Dealing with loops is a bit more complicated. For example, one can remove a transition from the loop to break it, but the choice of transition may involve a combinatorial enumeration to find the right transition. This type of counterexamples is not our focus in this paper and we leave it for future work. Interestingly, all of our case studies in Section 6 do not involve loop counterexamples.

5.3 Heuristics Considering Transition Relations

The simplest method to resolve a deadlock is to formulate a constraint imposing the existence of an outgoing transition from s_n. Since in this paper, our focus is on asynchronous systems, a transition is the execution of one of the processes. We propose two strategies for selecting a process to have an outgoing transition from a deadlock state.

Progress Heuristic. In this approach, we add a constraint stating that at least one of the proceses should have an outgoing transition from s_n. More formally, assume that the current topology includes i processes, where the read-set of each process has r variables, with domains $D_0, \dots, D_{r-1}$, and the write-set of each process includes w variables, with domains $D'_0, \cdots, D'_{w-1}$. Note that since the goal is to synthesize a symmetric program, all processes execute similarly according to the function T_p:

$$T_p : (\prod_{j \in [0, r-1]} D_j) \to (\prod_{j \in [0, w-1]} D'_j)$$

and function f is of type:

$$f : \mathbb{N} \to \left(S \to (\prod_{j \in [0, r-1]} D_j) \right)$$

Then, the constraint to be added to the SMT instance can be written as:

$$\forall j \in [0, w-1] : \exists val_j \in D'_j : \bigvee_{k \in [0, i)} \left(\left(f(k)(s_n), [val_0, val_1, \cdots, val_{w-1}] \right) \in T_p \right)$$

Example. Consider the counterexample mentioned in Section 5.1. Each process can read three Boolean variables and write to one Boolean variable and, hence, T_p is defined as follows:

$$T_p \ : \ \{\texttt{F}, \texttt{T}\} \times \{\texttt{F}, \texttt{T}\} \times \{\texttt{F}, \texttt{T}\} \rightarrow \{\texttt{F}, \texttt{T}\}$$

Note that for each process π_j, $f(j)$ returns $[x_{(j-1)}, x_j, x_{(j+1)}]$. In the counterexample we presented in the previous example, the last state where the deadlock happens is:

$$[x_0 = \texttt{T}, x_1 = \texttt{F}, x_2 = \texttt{T}, x_3 = \texttt{F}, x_4 = \texttt{F}]$$

Thus, we add the following constraint to the SMT instance:

$$\exists val \in \{\texttt{F}, \texttt{T}\} \ : \ \Big(([\texttt{F}, \texttt{T}, \texttt{F}], val) \in T_p \ \vee \ ([\texttt{T}, \texttt{F}, \texttt{T}], val) \in T_p \ \vee \ ([\texttt{F}, \texttt{T}, \texttt{F}], val) \in T_p \ \vee \\ ([\texttt{T}, \texttt{F}, \texttt{F}], val) \in T_p \ \vee \ ([\texttt{F}, \texttt{F}, \texttt{T}], val) \in T_p\Big)$$

In the above constraint, the jth clause imposes a constraint on T_p to have an outgoing transition considering the local state of the jth process. (Note that the first and third clauses are the same, and we just put them for clarity.)

Local LS Heuristic. As mentioned in Section 2, we focus on sets LS that can be locally defined, i.e., the set of legitimate states can be described as a conjunction over local legitimate states of processes. In this case, a deadlock can be resolved by checking the local state of each process and imposing a constraint to have an outgoing transition for at least one of those processes that are not in their local legitimate states.

Example. For the counterexample of one-bit maximal matching with 4 processes, the local set of legitimate states is already presented in the example below Definition 12, and the deadlock state is:

$$[x_0 = \texttt{T}, x_1 = \texttt{F}, x_2 = \texttt{T}, x_3 = \texttt{F}, x_4 = \texttt{F}]$$

For checking the local state of each process, we should first note the values of uninterpreted functions $match_i$ in this state:

$$[match_0 = l, match_1 = r, match_2 = l, match_3 = n, match_4 = l]$$

Processes π_0, π_1, π_3, and π_4 are not in a local legitimate state, and hence, the added constraint to the original SMT model will be as follows:

$$\exists val \in \{\texttt{F}, \texttt{T}\} \ : \ \Big(([\texttt{F}, \texttt{T}, \texttt{F}], val) \in T_p \ \vee \ ([\texttt{T}, \texttt{F}, \texttt{T}], val) \in T_p \ \vee \\ ([\texttt{T}, \texttt{F}, \texttt{F}], val) \in T_p \ \vee \ ([\texttt{F}, \texttt{F}, \texttt{T}], val) \in T_p\Big)$$

Note that although this method seems more efficient than the progress approach in terms of having shorter constraints, it has the drawback of missing some solutions that the previous approach can find. More specifically, for a process being in a legitimate local state in a deadlock state, it may be the case that taking a transition by this process leads to a state, from which its neighbors can take other transitions that finally leads to a legitimate state.

5.4 Heuristics Considering Uninterpreted Functions

Our second class of heuristics focus on uninterpreted functions, where we impose a constraint to change the interpretation function of at least one uninterpreted function in the deadlock state. Similar to the heuristics introduced for transition relations, we introduce two approaches for selecting at least one process to change the interpretation of its uninterpreted function. Because of the similarity to the previous heuristics, we skip the details of this heuristic.

6 Case Studies and Experimental Results

We used the model finder Alloy [15] and model checker NuSMV [7] to implement our counterexample-guided synthesis approach. Our experimental platform is an 2.9 GHz Intel Core i7 processor, with 16 GB of RAM. Our synthesis results are reported in Table 1.

6.1 Three Coloring

We consider the *three coloring problem* [14] on a ring, where each process π_i is associated with a variable c_i with domain $\{0, 1, 2\}$. Each value of the variable c_i represents a distinct color. A process can read and write its own variable. It can also read the variables of its neighbors. *LS* includes all states, where each process has a color different from its both neighbors. Thus, for a ring of 4 processes, *LS* is defined by the following predicate:

$$c_0(s) \neq c_1(s) \wedge c_1(s) \neq c_2(s) \ \wedge \ c_2(s) \neq c_3(s) \wedge c_3(s) \neq c_0(s)$$

Observe that the closure/deadlock-freedom cutoff point for this case study is $3^2 + 1 = 10$ and, hence, we need to synthesize a solution for 10 processes. The synthesis time reported in Table 1 is a bit smaller in the case of local *LS* heuristic, which is probably due to the smaller constraints added in this case. The resulting protocols for the two heuristics are different. Following is the one synthesized for the case of local LS:

$$\begin{array}{lrcl}
\pi_i: & (c_i = 2) \wedge (c_{(i+1)} = 2) \wedge (c_{(i-1)} \neq 0) & \rightarrow & c_i := 0 \\
& (c_i \neq 0) \wedge (c_{(i+1)} = 1) \wedge (c_{(i-1)} = 2) & \rightarrow & c_i := 0 \\
& (c_i = 1) \wedge (c_{(i+1)} = 1) \wedge (c_{(i-1)} \neq 2) & \rightarrow & c_i := 2 \\
& (c_i = 0) \wedge (c_{(i+1)} \neq 2) \wedge (c_{(i-1)} = 0) & \rightarrow & c_i := 2 \\
& (c_i \neq 1) \wedge (c_{(i+1)} = 2) \wedge (c_{(i-1)} = 0) & \rightarrow & c_i := 1
\end{array}$$

6.2 One-Bit Maximal Matching

This case study is the running example in this paper with cutoff point of $2^2 + 1 = 5$ processes. Note that using the heuristics considering transition relations, we could not synthesize a protocol for this problem (Alloy reports unsatisfiablity after adding the counterexample constraints). The interesting point about this case study is that the progress heuristic has better efficiency compared to the local *LS*. The reason may be due to the fact that the constraints added in the local *LS* heuristic are too restrictive, and hence, Alloy needs to search more in order to find a solution. The synthesized solutions using both heuristics are the same for this case study, where the transition relation is the following:

$$\begin{array}{lrcl}
\pi_i: & (x_i = \mathtt{F}) \wedge (x_{(i+1)} = \mathtt{F}) \wedge (x_{(i-1)} = \mathtt{F}) & \rightarrow & x_i := \mathtt{T} \\
& (x_i = \mathtt{T}) \wedge (x_{(i+1)} = \mathtt{T}) & \rightarrow & x_i := \mathtt{F}
\end{array}$$

Table 1 Results for parameterized synthesis.

Problem	cutoff #	Heuristic	Synthesis Time	Model Checking Time
Three Coloring	10	Local *LS*	7m 3sec	16 msec
Three Coloring	10	Progress	9m 5sec	16 msec
One-Bit MM	5	Local *LS*	1m 48sec	27 msec
One-Bit MM	5	Progress	1m 44sec	33 msec
Maximal Matching	10	Local *LS*	7m 59sec	36 msec
Maximal Matching	10	Progress	4m 57sec	37 msec
Maximal Independent Set	5	Local *LS*	10sec	18 msec

and the interpretation function for $match_i$ is the following:

$$
\begin{array}{llcl}
match_i: & (x_i = \mathtt{T}) \wedge (x_{(i+1)} = \mathtt{T}) \wedge (x_{(i-1)} = \mathtt{T}) & \mapsto & l \\
& (x_{(i+1)} = \mathtt{F}) \wedge (x_{(i-1)} = \mathtt{F}) & \mapsto & l \\
& (x_i = \mathtt{T}) \wedge (x_{(i+1)} = \mathtt{F}) \wedge (x_{(i-1)} = \mathtt{T}) & \mapsto & r \\
& (x_i = \mathtt{F}) \wedge (x_{(i+1)} = \mathtt{T}) & \mapsto & r \\
& (x_i = \mathtt{T}) \wedge (x_{(i+1)} = \mathtt{T}) \wedge (x_{(i-1)} = \mathtt{F}) & \mapsto & n \\
& (x_i = \mathtt{F}) \wedge (x_{(i+1)} = \mathtt{F}) \wedge (x_{(i-1)} = \mathtt{T}) & \mapsto & n
\end{array}
$$

6.3 Maximal Matching

In this case study, we used the same problem as in Section 6.2, but instead of using one Boolean variable for each process, we use a variable with three values $\{l, r, n\}$ and, hence, we do not need the uninterpreted functions anymore. The resulting protocols for the two heuristics are different. As an example, the synthesized protocol for the case of local *LS* is the following:

$$
\begin{array}{llcl}
\pi_i: & (x_i = n) \wedge (x_{(i+1)} = n) \wedge (x_{(i-1)} = n) & \rightarrow & x_i := r \\
& (x_i \neq r) \wedge (x_{(i+1)} \neq r) \wedge (x_{(i-1)} = l) & \rightarrow & x_i := r \\
& (x_i \neq n) \wedge (x_{(i+1)} = r) \wedge (x_{(i-1)} = l) & \rightarrow & x_i := n \\
& (x_i = n) \wedge (x_{(i-1)} = r) & \rightarrow & x_i := l \\
& (x_i = r) \wedge (x_{(i+1)} = r) \wedge (x_{(i-1)} \neq l) & \rightarrow & x_i := l
\end{array}
$$

6.4 Maximal Independent Set

An *independent set* in a graph is a subset of vertices in which no pair of vertices are adjacent. To synthesize a protocol that finds a maximal independent set, we consider a set of processes connected in a ring topology, where each process has a Boolean variable, the value of which shows whether or not it is included in the maximal independent set. The set of legitimate states include those states, where the processes whose variables have the true value form a maximal independent set. As an example, if c_i is the variable of the process π_i, then the set of legitimate states for the case of four processes is formulated by the following predicate:

$$
(c_0(s) = \mathtt{T} \wedge c_1(s) = \mathtt{F} \wedge c_2(s) = \mathtt{T} \wedge c_3(s) = \mathtt{F}) \vee (c_0(s) = \mathtt{F} \wedge c_1(s) = \mathtt{T} \wedge c_2(s) = \mathtt{F} \wedge c_3(s) = \mathtt{T})
$$

The point of this case study is that the resulting model for 4 processes worked for the size 5 as well, and hence, no counterexample is found. Therefore, the result for both heuristics is the same. The resulting protocol is the following:

$$\pi_i : \quad (x_i = \mathtt{F}) \wedge (x_{(i+1)} = \mathtt{F}) \wedge (x_{(i-1)} = \mathtt{F}) \quad \rightarrow \quad x_i := \mathtt{T}$$
$$(x_i = \mathtt{T}) \wedge (x_{(i+1)} = \mathtt{T}) \quad \rightarrow \quad x_i := \mathtt{F}$$

7 Related Work

Regarding the synthesis of self-stabilizing algorithms, one approach is to *add* self-stabilization to a given algorithm. In contrast to our approach, the technique proposed by Ebnenasir and Farahat [5] starts from a given non-stabilizing algorithm, it requires a more explicit specification of the legitimate states, and it is not complete, i.e., it may fail to find a solution even though there exists one. Klinkhammer and Ebnenasir show that adding strong convergence is NP-complete in the size of the state space, which itself is exponential in the size of variables of the protocol [18], and introduce a new method for adding self-stabilization that is complete, but otherwise has the same limitations as mentioned above [20]. For ring topologies, they have shown that parameterized verification of self-stabilization is undecidable in uni-directional rings [19], while the parameterized synthesis problem is undecidable in bi-directional rings, but surprisingly remains decidable in uni-directional rings [21]. Faghih and Bonakdarpour introduced an SMT-based synthesis technique for automatically synthesizing self-stabilizing systems [8, 9] that is complete and not based on existing non-stabilizing algorithms. An extension of this work [11] allows to symbolically specify the legitimate states as a set of requirements, and supports the synthesis of ideal-stabilizing systems.

While these approaches are promising and can automatically synthesize a number of well-known self-stabilizing systems, they all suffer from the problem of scalability, as the complexity of the problem increases exponentially in the number of processes. For example, all results reported by Faghih and Bonakdarpour [8, 9, 11] correspond to automatically synthesis of self-stabilizing systems with at most 5 processes. One way to address this scalability issue in synthesis is to use a counterexample-guided synthesis method, as it has been proposed for the completion of program sketches [25], for the lazy synthesis of reactive systems [13], and for the synthesis of Byzantine-resilient systems [1]. The latter approach also supports the synthesis of self-stabilizing systems, but counterexamples are only used to guide the encoding of Byzantine-resilience, and the approach is limited to synchronous systems. In all of these examples, a counterexample-guided approach can solve problems that are out of reach for existing approaches. In our work, we for the first time used counterexamples to guide synthesis for an increasing size of the topology, which allows us to scale the SMT-based synthesis of self-stabilizing algorithms to systems with up to 200 processes.

Finally, the problem of scalability in the number of processes can be solved once and for all by using a parameterized synthesis approach, as introduced by Jacobs and Bloem [16] for (non-stabilizing) reactive systems. The approach relies on cutoff results, similar to the ones we introduced in this work for closure and deadlock detection. Different techniques are introduced in [17] to improve scalability of this approach in the complexity of the specification, including the modular application of cutoff results in synthesis. An extension of the approach [1] also supports the parameterized synthesis of self-stabilizing systems, but only for synchronous systems, and not in all cases resulting in a completely symmetric system. Finally, Lazic et al. [22] propose a method for synthesizing parameterized fault-tolerant distributed algorithms. In contrast to our approach, synthesis is based on a sketch of an asynchronous threshold-based fault-tolerant distributed algorithm, and the goal is to find the right values for coefficients that may be missing in the guards.

8 Conclusion

In this paper, we proposed a new method for parameterized synthesis of self-stabilizing algorithms in symmetric rings using cutoff points. Furthermore, in order to scale the existing synthesis solutions [8, 9, 11, 12, 10] up to the cutoff point, we introduced an iterative loop of synthesis and verification guided by counterexamples. We demonstrated the effectiveness of our approach by synthesizing parameterized self-stabilizing protocols for well-known problems. For future, we plan to work on asymmetric and dynamic networks as well as the case, where the protocol is live in the set of legitimate states.

References

1. R. Bloem, N. Braud-Santoni, and S. Jacobs. Synthesis of self-stabilising and byzantine-resilient distributed systems. In *CAV*, pages 157–176, 2016.
2. S. Devismes, S. Tixeuil, and M. Yamashita. Weak vs. Self vs. Probabilistic Stabilization. In *ICDCS*, pages 681–688, 2008.
3. E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.
4. E. W. Dijkstra. A Belated Proof of Self-Stabilization. *Distributed Computing*, 1(1):5–6, 1986.
5. A. Ebnenasir and A. Farahat. A Lightweight Method for Automated Design of Convergence. In *IPDPS*, pages 219–230, 2011.
6. E. A. Emerson and K. S. Namjoshi. On Reasoning About Rings. *International Journal on Foundations of Computer Science.*, 14(4):527–550, 2003.
7. A. Cimatti et. al. Nusmv 2: An opensource tool for symbolic model checking. In *CAV*, pages 359–364, 2002.
8. F. Faghih and B. Bonakdarpour. SMT-based synthesis of distributed self-stabilizing systems. In *SSS*, pages 165–179, 2014.
9. F. Faghih and B. Bonakdarpour. SMT-based synthesis of distributed self-stabilizing systems. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 10(3):21, 2015.
10. F. Faghih and B. Bonakdarpour. ASSESS: A tool for automated synthesis of distributed self-stabilizing algorithms. In *SSS*, pages 219–233, 2017.
11. F. Faghih, B. Bonakdarpour, S. Tixeuil, and S. Kulkarni. Specification-based Synthesis of Distributed Self-Stabilizing Protocols. In *FORTE*, pages 124–141, 2016.
12. F. Faghih, B. Bonakdarpour, S. Tixeuil, and S. Kulkarni. Specification-based Synthesis of Distributed Self-Stabilizing Protocols. *Logical Methods in Computer Science*, To appear.
13. B. Finkbeiner and S. Jacobs. Lazy Synthesis. In *VMCAI*, 2012.
14. M. G. Gouda and H. B. Acharya. Nash Equilibria in Stabilizing Systems. In *SSS*, pages 311–324, 2009.
15. D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press Cambridge, 2012.
16. S. Jacobs and R. Bloem. Parameterized Synthesis. *Logical Methods in Computer Science*, 10(1), 2014.
17. A. Khalimov, S. Jacobs, and R. Bloem. Towards Efficient Parameterized Synthesis. In *VMCAI*, volume 7737 of *Lecture Notes in Computer Science*, pages 108–127. Springer, 2013.
18. A. Klinkhamer and A. Ebnenasir. On the Complexity of Adding Convergence. In *FSEN*, pages 17–33, 2013.
19. A. Klinkhamer and A. Ebnenasir. Verifying Livelock Freedom on Parameterized Rings and Chains. In *SSS*, pages 163–177, 2013.

20 A. Klinkhamer and A. Ebnenasir. Synthesizing Self-stabilization through Superposition and Backtracking. In *SSS*, pages 252–267, 2014.

21 A. Klinkhamer and A. Ebnenasir. Synthesizing Parameterized Self-stabilizing Rings with Constant-Space Processes. In *FSEN*, pages 100–115, 2017.

22 Marijana Lazic, Igor Konnov, Josef Widder, and Roderick Bloem. Synthesis of Distributed Algorithms with Parameterized Threshold Guards. In *OPODIS*, 2017.

23 F. Manne, M. Mjelde, L. Pilard, and S. Tixeuil. A new self-stabilizing maximal matching algorithm. *Theoretical Computer Science*, 410(14):1336–1345, 2009.

24 Kerry Raymond. A Tree-Based Algorithm for Distributed Mutual Exclusion. *ACM Transactions on Computer Systems*, 7(1):61–77, 1989.

25 Armando Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013.

Loosely-Stabilizing Leader Election with Polylogarithmic Convergence Time

Yuichi Sudo
Graduate School of Information Science and Technology, Osaka University, Japan
y-sudou@ist.osaka-u.ac.jp

Fukuhito Ooshita
Graduate School of Science and Technology, Nara Institute of Science and Technology, Japan
f-oosita@is.naist.jp

Hirotsugu Kakugawa
Graduate School of Information Science and Technology, Osaka University, Japan
kakugawa@ist.osaka-u.ac.jp

Toshimitsu Masuzawa
Graduate School of Information Science and Technology, Osaka University, Japan
masuzawa@ist.osaka-u.ac.jp

Ajoy K. Datta
Department of Computer Science, University of Nevada, Las Vegas, USA
ajoy.datta@unlv.edu

Lawrence L. Larmore
Department of Computer Science, University of Nevada, Las Vegas, USA
lawrence.larmore@unlv.edu

Abstract

A loosely-stabilizing leader election protocol with polylogarithmic convergence time in the population protocol model is presented in this paper. In the population protocol model, which is a common abstract model of mobile sensor networks, it is known to be impossible to design a self-stabilizing leader election protocol. Thus, in our prior work, we introduced the concept of loose-stabilization, which is weaker than self-stabilization but has similar advantage as self-stabilization in practice. Following this work, several loosely-stabilizing leader election protocols are presented. The loosely-stabilizing leader election guarantees that, starting from an arbitrary configuration, the system reaches a safe configuration with a single leader within a relatively short time, and keeps the unique leader for an sufficiently long time thereafter. The convergence times of all the existing loosely-stabilizing protocols, i.e., the expected time to reach a safe configuration, are polynomial in n where n is the number of nodes (while the holding times to keep the unique leader are exponential in n). In this paper, a loosely-stabilizing protocol with polylogarithmic convergence time is presented. Its holding time is not exponential, but arbitrarily large polynomial in n.

2012 ACM Subject Classification Theory of computation → Self-organization

Keywords and phrases Loose-stabilization, Population protocols, and Leader election

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.30

Acknowledgements This work was supported by JSPS KAKENHI Grant Numbers 17K19977, 16K00018, and 18K18000, and Japan Science and Technology Agency(JST) SICORP.

© Yuichi Sudo, Fukuhito Ooshita, Hirotsugu Kakugawa, Toshimitsu Masuzawa, Ajoy K. Datta, and Lawrence L. Larmore;
licensed under Creative Commons License CC-BY

22nd International Conference on Principles of Distributed Systems (OPODIS 2018).
Editors: Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira; Article No. 30; pp. 30:1–30:16

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

1 Introduction

We consider the *population protocol* (PP) model [3] in this paper. A network called *population* consists of a large number of finite-state automata, called *agents*. Agents often make *interactions* (i.e., pairwise communication) each between a pair of agents by which they update their states. The interactions are opportunistic, that is, they are unknown and unpredictable (or predictable only with probability). Agents are strongly anonymous: they do not have identifiers and they cannot distinguish their neighbors with the same states. As with the majority of studies on population protocols, we assume that the network of agents is a complete graph, and that the scheduler selects an interacting pair of agents at each step uniformly at random.

In this paper, we focus on the problem of self-stabilizing leader election (SS-LE), which is one of the most important and well-studied problems in the PP model. Self-stabilizing leader election requires that starting from any configuration, a population reaches a safe configuration in which exactly one leader exists; and after that, the population keeps that leader forever. These requirements guarantee excellent tolerance against any finite number of transient faults. Since many protocols (whether self-stabilizing or non-stabilizing) in the literature work with the assumption that a unique leader exists [3, 4, 5], SS-LE is a key to improving the fault-tolerance of the PP model itself. However, SS-LE is strictly impossible in the PP model: no protocol can solve SS-LE unless every agent in the population knows the exact size of the population (i.e., the number of agents) [4]. This impossibility comes from a simple partitioning argument[1]. Thus, most of the studies extend (i.e., strengthen) the PP model to circumvent the impossibility. One approach of studies [7, 13, 18] assumes that every agent knows the exact value of n and focuses on the space complexity to solve SS-LE. Another approach [9, 6, 8] solves SS-LE by using *oracles*, which tell every agent whether or not there exists an agent in a leader-state.

To solve SS-LE in a more practical way, our previous work [14] introduces the concept of *loose-stabilization*, which relaxes the closure requirement of self-stabilization but keeps its advantage in practice. Specifically, starting from any initial configuration, the population must reach a safe configuration within a relatively short time; after that, the specification of the problem (the unique leader for leader election) must be sustained for a sufficiently long time, though not necessarily forever. In [14], we gave a loosely-stabilizing leader election (LS-LE) protocol assuming that every agent knows a common upper bound N of n. This protocol is practically equivalent to an SS-LE protocol since it maintains the unique leader for exponential time in n (that is, practically forever) after reaching a safe configuration within $O(N \log N)$ *parallel time*, which we will define later. The assumption that we can use an upper bound N of n is practical because the protocol works correctly even if we make a large overestimation of n, such as $N = 10n$. Recently, Izumi [11] give a method which improves the convergence time of this protocol to linear time, i.e., $O(N)$. In [15, 16, 17], LS-LE protocols are presented for a population where some pairs of agents may not have interactions, i.e., the interaction graph is not complete.

---

[1] Assume that a SS-LE protocol P works without knowledge of exact size n of the population. Then, there must exist $n_1, n_2, (n_1 < n_2)$ such that P works correctly both when $n = n_1$ and when $n = n_2$. Consider an execution of P on the population with size n_2 which starts from a safe configuration where exactly one leader exists. If some $n_2 - n_1$ agents including the unique leader do not interact for a sufficiently long time, the rest of the population consisting of the other n_1 agents must create a new leader to satisfy the convergence requirement of self-stabilization. However, this creation results in two leaders in the population, violating the closure requirement of self-stabilization.

Leader election is easily solved in linear parallel time in the PP model if one does not stick to self- or loose- stabilization. When we design non-stabilizing protocols, we can assume that all agents are in a specific state in the initial configuration. A non-stabilizing leader election protocol was first presented in [3], which elects a unique leader within $O(n)$ parallel time. Recently, Alistarh and Gelashvili [1] give a non-stabilizing leader election protocol with polylogarithmic parallel time. Starting from a specific initial configuration, their protocol elects a unique leader within $O(\log^3 n)$ parallel time, with the assumption that all agents share a common integer $m = \Theta(\log^3 n)$. More recently, Gąsieniec and Staehowiak [10] give an algorithm which converges in $O(\log^2 n)$ parallel time. (The space complexity of the algorithm [10] is also surprisingly small, i.e., $O(\log\log\log n)$ bits.) The key strategies used in these papers to achieve polylogarithmic time are interesting, but we can not utilize them for our purpose because both of them critically depend on the assumption that all agents have the same initial state.

A number of results in the PP model assume the uniformly random scheduler, that is, a pair of agents is chosen uniformly at random to interact at each step [3, 5, 11, 14, 2, 10, 1]. This assumption has been used mainly for evaluating the time complexity of protocols. We also adopt this assumption because the measure of time is crucial in the concept of loose-stabilization. In the PP model, time complexities, such as convergence time and holding time, are often evaluated in *parallel time*, which is defined as the expected number of interactions or steps, divided by n (i.e., the number of agents). This is a natural measure of time because, in practice interactions typically occur in parallel in the population.

1.1 Our Contribution

We now present an LS-LE protocol with polylogarithmic convergence time and polynomial holding time. Here, and for the remainder of this section, when we discuss time complexity, we shall always presume parallel computation. To the best of our knowledge, all previously published LS-LE protocols have at least linear convergence time, and exponential holding time, as shown in Table 1. Our protocol P_{PL} breaks through the barrier of linear convergence time. There is a convergence time/holding time trade-off. Given a parameter $c \geq 1$ and an upper bound N of n, our protocol converges within $O(c\log^3 N)$ time, and has $\Omega(cn^{10c})$ holding time. Although our expected holding time is not exponential in N, it grows as an exponential function of c. Also, the convergence time of P_{PL} does not suffer much from large overestimation of n; it is always $O(c\log^3 n)$ as long as N is polynomial in n. It is worth mentioning that P_{PL} has small space complexity. Each agent needs only $O(\log\log N)$ bits of memory to store all variables of P_{PL}. We can say that this is small space when we consider that any self-stabilizing leader election protocol, which requires knowledge of n, needs the space of at least $\lceil \log n \rceil$ bits [7]. These performances of our protocol cannot be obtained if we require exponential holding time: it is proven by Izumi [11] that any LS-LE protocol whose holding time is exponential requires $\Omega(N)$ convergence time and $\Omega(\log N)$-bit space at each agent.

We obtain a useful tool when analyzing the convergence and holding time of P_{PL}. Let `var` be a variable of some algorithm. Consider that two agents interact and the values of `var` in the two agents change from x and y to x' and y'. We call `var` a *propagating variable* if both $x' \geq \max(x-1, y-1, 0)$ and $y' \geq \max(x-1, y-1, 0)$ are always guaranteed. To the best of our knowledge, all loosely-stabilizing protocol (and possibly some non-stabilizing protocols) use propagating variables. Let z and Δ be any integers. If some agent has value $z + \Delta$ for a propagating variable `var` and Δ is sufficiently large, the propagating property of `var` guarantees that all the agents obtain values larger than z in `var` in a short time;

Table 1 Self/Loosely-stabilizing leader election in the PP model (shown in parallel time).

Protocol	Type	Knowledge	Convergence Time	Holding Time	Agent Space (bits)
[14]	loose-stab.	N	$O(N \log N)$	$\Omega(e^N)$	$O(\log N)$
[11]	loose-stab.	N	$O(N)$	$\Omega(e^N)$	$O(\log N)$
P_{PL}	loose-stab.	N	$O(c \log^3 N)$	$\Omega(cn^{10c})$	$O(\log \log N)$
[7]	self-stab.	(exact) n	polynomial	∞	$O(\log n)$

specifically, in $O(\log n)$ time with high probability. The interesting question is how large this Δ should be. In the analysis of [14], a trivially sufficient value $\Delta = \Theta(n)$ is used. However, this linear value is useless for designing a protocol with polylogarithmic convergence time. In this paper, we prove that $\Delta = \Theta(\log n)$ is sufficient to propagate of values larger than z to the whole population (Lemma 6). This result may seem trivial for experts at the first glance, however, it is not trivial. This is because, whereas every agent participates in each interaction of an execution with probability $2/n$, the probability that a *virtual agent* (defined later) participates in each interaction may not be equal to $2/n$. We prove $\Delta = \Theta(\log n)$ by using three different kinds of random variables and bounding Δ by the sum of them. As we will see later, this result is very helpful in the analysis of the behavior of a protocol with propagating variables.

2 Preliminaries

In this section, we describe our model of computation. We denote the set of integers $\{z \in \mathbb{N} \mid x \le z \le y\}$ by $[x, y]$, and denote the nth harmonic number by $H_n = \sum_{k=1}^{n} \frac{1}{k}$. We write the natural logarithm of x as $\ln x$; we indicate the base of other logarithms of x, such as $\log_2 x$.

A *population* is a network consisting of *agents*. We denote the set of all the agents by V and let $n = |V|$. We assume that a population is complete graph, thus every pair of agents (u, v) can interact, where u serves as the *initiator* and v serves as the *responder* of the interaction.

A *protocol* $P(Q, Y, T, \pi_{out})$ consists of a finite set Q of states, a finite set Y of output symbols, a transition function $T : Q \times Q \to Q \times Q$, and an output function $\pi_{out} : Q \to Y$. When two agents interact, T determines their next states according to their current states. The *output* of an agent is determined by π_{out}: the output of an agent in state q is $\pi_{out}(q)$.

A *configuration* is a mapping $C : V \to Q$ that specifies the states of all the agents. We denote the set of all configurations of protocol P by $\mathcal{C}_{\mathrm{all}}(P)$. We say that a configuration C changes to C' by the interaction $e = (u, v)$, denoted by $C \xrightarrow{e} C'$, if $(C'(u), C'(v)) = T(C(u), C(v))$ and $C'(w) = C(w)$ for all $w \in V \setminus \{u, v\}$.

A *schedule* $\gamma = \gamma_0, \gamma_1, \dots = (u_0, v_0), (u_1, v_1), \dots$ is a sequence of interactions. A schedule determines which interaction occurs at each time, i.e., interaction γ_t happens at time t under schedule γ. In particular, we consider a *uniformly random scheduler* $\Gamma = \Gamma_0, \Gamma_1, \dots$ in this paper: each Γ_t is a random variable such that $\Pr(\Gamma_t = (u, v)) = \frac{1}{n(n-1)}$ for any $t \ge 0$ and any distinct $u, v \in V$. Note that we use capital letter Γ for this uniform random scheduler while we refer a deterministic schedule with a lower case such as γ. Given an initial configuration C_0 and a schedule γ, the *execution* of protocol P is defined as $\Xi_P(C_0, \gamma) = C_0, C_1, \dots$ such that $C_t \xrightarrow{\gamma_t} C_{t+1}$ for all $t \ge 0$. Note that the execution $\Xi_P(C_0, \Gamma) = C_0, C_1, \dots$ under the uniformly random scheduler is a sequence of configurations where each C_i is a random variable. For a schedule $\gamma = \gamma_0, \gamma_1, \dots$ and any $t \ge 0$, we say that agent $v \in V$ *participates* in γ_t if v is either the initiator or the responder of γ_t.

The leader election problem requires that every agent should output L or F which means "leader" or "follower" respectively. The specification of leader election, denoted by LE, requires that there exists one agent v such that v is always a leader and all other agents are always followers throughout an execution. We define $\mathrm{EIH}_P(C, LE)$ as the expected number of interactions during which an execution $\Xi_P(C,\Gamma)$ starting from a configuration $C \in \mathcal{C}_{\mathrm{all}}(P)$ keeps LE (i.e., the expected number of interactions until $\Xi_P(C,\Gamma)$ deviates from LE). For any set $\mathcal{S} \subseteq \mathcal{C}_{\mathrm{all}}(P)$ of configurations, we also define $\mathrm{EIC}_P(C,\mathcal{S})$ as the expected number of interactions required for the population to enter a configuration in $\mathcal{S}$ in an execution $\Xi_P(C,\Gamma)$ starting from a configuration $C \in \mathcal{C}_{\mathrm{all}}(P)$. The notation EIH (resp. EIC) stands for the Expected number of Interactions to Hold (resp. Converge).

▶ **Definition 1** (Loose-stabilizing leader election [14])**.** Protocol $P(Q, Y, T, \pi_{out})$ is an (α, β)-loosely-stabilizing leader election protocol if there exists a set $\mathcal{S}$ of configurations satisfying the following two inequalities:

$$\max_{C \in \mathcal{C}_{\mathrm{all}}(P)} \mathrm{EIC}_P(C,\mathcal{S}) \leq \alpha \quad \text{and} \quad \min_{C \in \mathcal{S}} \mathrm{EIH}_P(C, LE) \geq \beta.$$

We call a configuration in $\mathcal{S}$ in the above definition *a safe configuration* of P. In terms of parallel time, (α, β)-loosely-stabilizing protocol leader election protocol P reaches a safe configuration within α/n parallel time in expectation and keeps the unique leader for β/n parallel time in expectation thereafter. We call α/n and β/n *the expected holding time* and *the expected convergence time* of P.

Throughout the paper, we will use the following three variants of Chernoff bounds.

▶ **Lemma 2** ([12], Theorems 4.4, 4.5)**.** *Let $X_1, \ldots, X_s$ be independent Poisson trials, and let $X = \sum_{i=1}^{s} X_i$. Then*

$$\forall \delta,\ 0 \leq \delta \leq 1:\ \Pr(X \geq (1+\delta)\mathbf{E}[X]) \leq e^{-\delta^2 \mathbf{E}[X]/3}, \tag{1}$$

$$\forall R \geq 6\mathbf{E}[X]:\ \Pr(X \geq R) \leq 2^{-R}, \tag{2}$$

$$\forall \delta,\ 0 < \delta < 1:\ \Pr(X \leq (1-\delta)\mathbf{E}[X]) \leq e^{-\delta^2 \mathbf{E}[X]/2}. \tag{3}$$

3 Protocol P_{PL}

We give a loosely-stabilizing leader election protocol P_{PL}. This protocol uses a given upper bound N on n and has a parameter $c \geq 1$ by which we can adjust the expected convergence time and the expected holding time. As mentioned above, time complexity is measured as *parallel time*, which is defined as the number of interactions divided by n. The expected convergence time of P_{PL} is $O(c \log n \cdot \log^2 N) \subseteq O(c \log^3 N)$ and the expected holding time is $\Omega(cn^{10c+1})$. Thus, we achieve loosely-stabilizing leader election with polylogarithmic convergence time and polynomial holding time by setting $c = \Theta(1)$. For example, assigning $c = 10$ gives $O(\log^3 N)$ convergence time and $\Omega(n^{100})$ holding time.

The pseudo code of P_{PL} is given as Algorithm 1. Each agent has five variables: $\texttt{leader} \in \{\top, \bot\}$, $\texttt{shield} \in \{\top, \bot\}$, $\texttt{virus} \in [0, t_{\mathrm{virus}}]$, $\texttt{timer}_L \in [0, t_{\max}]$, and $\texttt{timer}_I \in [0, t_{\mathrm{emit}}]$. The first two variables leader and shield are Boolean variables: $v.\texttt{leader} = \top$ means that v is a leader, $v.\texttt{shield} = \top$ means that v is *shielded*, which will be explained later. The next three variables virus, $\texttt{timer}_L$, and $\texttt{timer}_I$ are count-down timers where their maximum values are $t_{\mathrm{virus}} = 60\lceil \ln N \rceil$, $t_{\max} = 12c \cdot t_{\mathrm{virus}} \lceil \ln N \rceil$, and $t_{\mathrm{emit}} = 12c \cdot t_{\mathrm{virus}} \lceil \ln N \rceil$, respectively (Note that $t_{\max} = t_{\mathrm{emit}}$). The output function π_{out} is defined as follows: an agent with $\texttt{leader} = \top$ (resp. $\texttt{leader} = \bot$) outputs L (resp. F).

Algorithm 1 P_{PL}.

Constants:

c, N // given parameters. $N \geq n$ is guaranteed.

$t_{\mathrm{virus}} = 60\lceil \ln N \rceil$

$t_{\max} = 12c \cdot t_{\mathrm{virus}} \lceil \ln N \rceil$

$t_{\mathrm{emit}} = 12c \cdot t_{\mathrm{virus}} \lceil \ln N \rceil$

Variables of each agent:

$\mathtt{leader} \in \{\top, \bot\}$, $\mathtt{shield} \in \{\top, \bot\}$,

$\mathtt{timer}_L \in [0, t_{\max}]$, $\mathtt{virus} \in [0, t_{\mathrm{virus}}]$, $\mathtt{timer}_I \in [0, t_{\mathrm{emit}}]$,

Output function π_{out}:

if $v.\mathtt{leader} = \top$ holds, then the output of agent v is L, otherwise F.

Interaction between initiator a_0 and responder a_1:

1: $a_0.\mathtt{timer}_L \leftarrow a_1.\mathtt{timer}_L \leftarrow \max(a_0.\mathtt{timer}_L - 1,\ a_1.\mathtt{timer}_L - 1, 0)$
2: **for** $i \in \{0,1\}$ such that $a_i.\mathtt{timer}_L = 0$ **do** $a_i.\mathtt{leader} \leftarrow \top$ **endfor**
3: **if** $\exists i \in \{0,1\} : a_i.\mathtt{leader} = \top$ **then** $a_0.\mathtt{timer}_L \leftarrow a_1.\mathtt{timer}_L \leftarrow t_{\max}$ **endif**

4: $a_0.\mathtt{virus} \leftarrow a_1.\mathtt{virus} \leftarrow \max(a_0.\mathtt{virus} - 1,\ a_1.\mathtt{virus} - 1, 0)$
5: **for** $i \in \{0,1\}$ such that $\neg a_i.\mathtt{shield} \wedge (a_i.\mathtt{virus} > 0)$ **do** $a_i.\mathtt{leader} \leftarrow \bot$ **endfor**

6: **for** $i \in \{0,1\}$ **do** $a_i.\mathtt{timer}_I \leftarrow \max(a_i.\mathtt{timer}_I - 1, 0)$ **endfor**
7: **if** $a_0.\mathtt{timer}_I = 0 \wedge a_0.\mathtt{leader} = \top$ **then** $(a_0.\mathtt{virus}, a_0.\mathtt{shield}) \leftarrow (t_{\mathrm{virus}}, \top)$ **endif**
8: **if** $a_1.\mathtt{timer}_I = 0 \wedge a_1.\mathtt{leader} = \top$ **then** $a_1.\mathtt{shield} \leftarrow \bot$ **endif**
9: **for** $i \in \{0,1\}$ such that $a_i.\mathtt{timer}_I = 0$ **do** $a_i.\mathtt{timer}_I \leftarrow t_{\mathrm{emit}}$ **endfor**

Protocol P_{PL} consists of a *timeout mechanism* (Lines 1-3) and a *virus-war mechanism* (Lines 4-9). By using variable $\mathtt{timer}_L$, the timeout mechanism creates a leader when no leader exists in the population. By using variables $\mathtt{timer}_I$, $\mathtt{virus}$, and $\mathtt{shield}$, the virus-war mechanism reduces the number of leaders if there are two or more leaders.

The timeout mechanism of P_{PL} (Lines 1-3) is almost the same as that of the protocol given in [14]. This mechanism uses a leader-timer $\mathtt{timer}_L$, which indicates the possibility of existence of a leader. A leader agent always keeps $\mathtt{timer}_L = t_{\max}$, and resets the leader-timer of the other agent to $t_{\max}$ every time it interacts with a non-leader agent (Line 3). We call this operation *timer reset*. When two non-leaders interact, we take the larger timer value of the two agents, decrease it by one, and substitute the decreased value into the leader-timers of both agents (Line 1). We call this operation *larger value propagation*. When the leader-timer of a non-leader decreases to zero, it suspects that no leader exists in the population, and it becomes a new leader (Line 2). We call this event *timeout*. This mechanism works well for the following reasons: (i) the timeout rarely happens when the population has a leader because the leader-timers of all agents have large values thanks to the timer reset and the larger value propagation, (ii) when no leader exists in the population, a timeout happens and a new leader is created within a short time because there is no possibility of a timer reset. It is proven in [14] that, if $t_{\max} = \Omega(n)$, the timeout rarely happens in the population with at least one leader. In the next section, we show that, with a high probability, the timeout does not happen for a long time when a leader exists, even if the maximum value $t_{\max}$ is polylogarithmic in N, specifically, $t_{\max} = \Theta(\log^2 N)$.

The basic idea of the virus-war mechanism is first presented in [15]. P_{PL} uses this idea, but implements it in a considerably different way in order to reduce the number of leaders to

one within a polylogarithmic parallel time. In the virus-war mechanism (Lines 4-9), every leader tries to kill other leaders by using viruses and become the unique leader. We say that agent v has a virus if $v.\mathtt{virus} > 0$, and that v is shielded if $v.\mathtt{shield} = \top$. As we will see later, a virus is propagated to the whole population by interactions and kills leaders that are not shielded. Every agent has an individual timer $\mathtt{timer}_I$ to create a new virus periodically. This timer is decreased by one every time that agent participates in an interaction (Line 6). When the individual timer of a leader reaches zero at an interaction, its fate differs according to its role in the interaction, initiator or responder. If the agent is an initiator, it succeeds in creating a new virus and becomes shielded, that is, $\mathtt{virus} \leftarrow t_{\mathrm{virus}}$ and $\mathtt{shield} \leftarrow \top$ (Line 7). If it is a responder, it becomes unshielded i.e., $\mathtt{shield} \leftarrow \bot$ (Line 8). Thereafter, the individual timer is reset to the maximum value t_{emit} in both cases (Line 9). A virus spreads by interactions (Line 4). A leader is killed and becomes a non-leader if it catches a virus when it is not shielded (Line 5). The value of $\mathtt{virus}$ in an agent corresponds to the TTL (time to live) of the virus that the agent carries. Since it decreases in the larger value propagation fashion, viruses eventually disappear from the population and no virus exists in the following execution until a new virus is created by some leader.

When there are multiple leaders, the virus-war mechanism elects exactly one leader within a short time. Since every agent participates in any interaction Γ_t with probability $2/n$, the individual timer of an agent reaches zero within $O(nt_{\mathrm{emit}})$ interactions with high probability. Therefore, some leader creates a new virus (and becomes shielded) by executing Line 7 within $O(nt_{\mathrm{emit}})$ interactions in expectation and approximately half of leaders are killed by the virus. Therefore, the number of shielded leaders is approximately halved for every $O(nt_{\mathrm{emit}})$ interactions. This rough and intuitive analysis explains why $O(nt_{\mathrm{emit}} \log n) \subseteq O(cn \log^3 N)$ interactions are sufficient to elect a unique leader. On the other hand, we must consider the risk of *suicide*, i.e., the event where a single leader creates a new virus and then becomes unshielded before the virus disappears from the population. This event causes the unique leader to be killed by the virus. A long holding time cannot be achieved if suicides are frequent. Suicide of a single leader v occurs only when a leader v first executes Line 7, starting a new virus while shielding itself, and then later executes Line 8 while the virus is still present, causing itself to be killed shortly thereafter. We can ensure that the frequency of that sequence of events is extremely small by assigning a sufficiently large value to t_{emit} compared to t_{virus}, as we discuss carefully in the next section.

4 Analysis of Convergence and Holding Time

In this section, we prove that P_{PL} is a $(O(cn \log n \cdot \log^2 N), \Omega(cn^{10c+1}))$-loosely-stabilizing leader election protocol. Define the set $\mathcal{S}$ of safe configurations as follows:

$$\begin{aligned}
\mathcal{L}_i &= \{C \in \mathcal{C}_{\mathrm{all}}(P_{\mathrm{PL}}) \mid 1 \le |\{v \in V \mid C(v).\mathtt{leader}\}| \le i\}, \text{ for } i = 1, 2, \dots, n \\
\mathcal{G}_{\mathrm{half}} &= \{C \in \mathcal{C}_{\mathrm{all}}(P_{\mathrm{PL}}) \mid \forall v \in V : C(v).\mathtt{timer}_L \ge t_{\max}/2\}, \\
\mathcal{L}_{\mathrm{safe}} &= \{C \in \mathcal{C}_{\mathrm{all}}(P_{\mathrm{PL}}) \mid \exists v \in V : C(v).\mathtt{leader} \land C(v).\mathtt{shield} \land C(v).\mathtt{timer}_I \ge t_{\mathrm{emit}}/2\}, \\
\mathcal{V}_{\mathrm{clean}} &= \{C \in \mathcal{C}_{\mathrm{all}}(P_{\mathrm{PL}}) \mid \forall v \in V : C(v).\mathtt{virus} = 0\}. \\
\mathcal{S} &= \mathcal{L}_1 \cap \mathcal{G}_{\mathrm{half}} \cap (\mathcal{L}_{\mathrm{safe}} \cup \mathcal{V}_{\mathrm{clean}}),
\end{aligned}$$

We need to prove the following two equalities:

$$\max_{C \in \mathcal{C}_{\mathrm{all}}(P_{\mathrm{PL}})} \mathrm{EIC}_{P_{\mathrm{PL}}}(C, \mathcal{S}) = O(cn \log n \cdot \log^2 N), \tag{4}$$

$$\min_{C \in \mathcal{S}} \mathrm{EIH}_{P_{\mathrm{PL}}}(C, LE) = \Omega\left(cn^{10c+1}\right). \tag{5}$$

Note that $C \in \mathcal{L}_i$ requires that the population in configuration C has at least one leader but does not have more than i leaders. For example, $\mathcal{L}_1$ is the set of configuration where exactly one leader exists in the population and $\mathcal{L}_n$ is the set of configurations where at least one leader exists in the population. Clearly, $\mathcal{L}_1 \subset \mathcal{L}_2 \subset \cdots \subset \mathcal{L}_n$ holds.

In the remainder of this section, we first introduce analytic tools (the notions of epidemic and virtual agents) in Section 4.1. Expected holding time and expected convergence time are analyzed in Sections 4.2 and 4.3, respectively. Since we evaluate the expected convergence time and the expected holding time only asymptotically, it suffices to assume that n is sufficiently large. Specifically, we assume $n \geq 55$; we then have $\lceil \ln N \rceil \geq 5$.

4.1 Tools

The goal of this Section is, intuitively, to prove $\Delta = \Theta(\log n)$ where Δ is the integer introduced at the end of Section 1.1. First, we review the notions of *epidemic* and *virtual agents* presented in [5] and [14] respectively. Most of the definitions in Section 4.1 are borrowed from [14].

By an abuse of notation, we will identify an interaction (u, v) with the set $\{u, v\}$ whenever convenient. Let $\gamma = \gamma_0, \gamma_1, \ldots$ be an infinite sequence of interactions and let r be an agent in V. The *epidemic function* $I_{r,\gamma} : [0, \infty) \to 2^V$ is defined as follows: $I_{r,\gamma}(0) = \{r\}$, and for $t = 1, 2, \ldots$, $I_{r,\gamma}(t) = I_{r,\gamma}(t-1) \cup \gamma_{t-1}$ if $I_{r,\gamma}(t-1) \cap \gamma_{t-1} \neq \emptyset$; otherwise, $I_{r,\gamma}(t) = I_{r,\gamma}(t-1)$. We say that v is *infected* at time t if $v \in I_{r,\gamma}(t)$ in the epidemic starting from agent r under γ. At time 0, only r is infected; at later steps, an agent becomes infected if it interacts with an infected agent. Once an agent becomes infected, it remains infected thereafter. We define the *infection time* $t_{r,\gamma}(v)$ of agent $v \in V$ to be $\min\{t \geq 0 \mid v \in I_{r,\gamma}(t+1)\}$ if $v \neq r$, and we let $t_{r,\gamma}(r) = -1$. Note that every agent $v \neq r$ is infected by the interaction $\gamma_{t_{r,\gamma}(v)}$.

We now define the *virtual agent* $VA_{r,\gamma}(v)$ of each agent $v \in V$. We assume that all agents eventually become infected, that is, $I_{r,\gamma}(t') = V$ holds for some t'. The virtual agent $VA_{r,\gamma}(v)$ is not defined if no such t' exists.[2] Let v be any agent other than r. We define the *parent* of v as the agent that infects v at time $t_{r,\gamma}(v)$. This parent-child relation defines a spanning tree of G rooted at r. In this tree, we call the unique path from r to v i.e., $v_0(= r), v_1, v_2, \ldots, v_k(= v)$, the *infection path* of v. The virtual agent $VA_{r,\gamma}(v)$ is a virtual entity that migrates from r to v through this infection path. At the beginning, $VA_{r,\gamma}(v)$ stays at r. For every $i \in [0, k-1]$, it migrates from v_i to v_{i+1} at time $t_{r,\gamma}(v_{i+1})$ After reaching $v_k = v$, the virtual agent remains at v. For $t \geq 0$, we say that virtual agent $VA_{r,\gamma}(v)$ *participates* in γ_t if, at time t, $VA_{r,\gamma}(v)$ is at one of the two agents participating in γ_t. For any $t \geq 1$, we define $VI_{r,\gamma}(v, t)$ as the number of interactions in $\gamma_0, \gamma_1, \ldots, \gamma_{t-1}$ that $VA_{r,\gamma}(v)$ participates in. Formally, we define $VI_{r,\gamma}(v, t) = |\{j \in [0, t-1] \mid (v \in \gamma_j \land j \geq t_{r,\gamma}(v)) \lor (\exists i \in [0, k-1] : v_i \in \gamma_j \land t_{r,\gamma}(v_i) \leq j < t_{r,\gamma}(v_{i+1}))\}|$.

Consider an execution $\Xi_P(C_0, \gamma) = C_0, C_1, \ldots$ of some protocol P that has a propagating variable `var`. Larger value propagation guarantees that the virtual agent $VA_{r,\gamma}(v)$ brings a large value of `var` from r to v when it reaches v through the infection path if r has a sufficiently large value of `var` in C_0 and virtual agent $VA_{r,\gamma}(v)$ participates in sufficiently few interactions in $\gamma_0, \gamma_1, \ldots, \gamma_{t_{r,\gamma}(v)}$. This property is formalized as the following trivial lemma.

▶ **Lemma 3.** *Let $\gamma = \gamma_0, \gamma_1, \ldots$ be a schedule, P be a protocol, and* `var` *be a propagating variable of P. Let $C_0 \in \mathcal{C}_{\mathrm{all}}(P)$ and $\Xi_P(C_0, \gamma) = C_0, C_1, \ldots$. Then $v \in I_{r,\gamma}(t) \Rightarrow C_t(v).\mathtt{var} \geq C_0(r).\mathtt{var} - VI_{r,\gamma}(v, t)$ for any agents $r, v \in V$ and any integer $t \geq 1$.*

[2] Such t' exits with probability 1 when the uniformly random scheduler Γ is given.

Angluin et al. [5] prove that the epidemic from any agent $r \in V$ finishes (i.e., all agents are infected) within $\Theta(n \log n)$ interactions with high probability. Furthermore, our previous work [14] gives a concrete lower bound on the probability that the epidemic finishes within a given number of interactions as follows:

▶ **Lemma 4** ([14])**.** *Let $r \in V$. For any integer $t \geq 1$, $\Pr(I_{r,\Gamma}(2t) \neq V) \leq ne^{-t/n}$.*

From Lemmas 3 and 4, we can achieve the goal of this section by giving a sufficiently tight lower bound of $VI_{r,\gamma}(v,t)$ with high probability. It would be easy if the probabilities of $VA_{r,\gamma}(v) \in \Gamma_t$ for distinct t were independent of each other, and equal to $2/n$, like the probability of $v \in \Gamma_t$. Then, the simple Chernoff bound would give a tight lower bound with high probability. However, $\Pr(VA_{r,\gamma}(v) \in \Gamma_t) = 2/n$ does not hold in general. This is because $VA_{r,\gamma}(v)$ at time t is determined not only by the preceding interactions $\Gamma_0, \Gamma_1, \dots, \Gamma_{t-1}$ but also by the subsequent interactions. $\Gamma_t, \Gamma_{t+1}, \dots$. Therefore, we need a careful analysis. The following lemma is one of the main contributions in this paper; it enables us to bound $VI_{r,\Gamma}(v,t)$ probabilistically to a logarithmic order when t is logarithmic in n.

▶ **Lemma 5.** *The following two inequalities hold for any $r, v \in V$, $t \geq 1$, $d \geq 3$, $d' \geq 3$, and $d'' \geq 6$:*

$$\Pr\left(VI_{r,\Gamma}(v,t) \geq 4d\lceil \log_2 n \rceil + 2d'H_n + \frac{4d''t}{n} \;\middle|\; I_{r,\Gamma}(t) = V \right) \leq n^{-d} + n^{-d'} + e^{-\frac{2d''t}{n}},$$

$$\Pr\left(VI_{r,\Gamma}(v,t) \geq 4d\lceil \log_2 n \rceil + 2d'H_n + \frac{8t}{n} \;\middle|\; I_{r,\Gamma}(t) = V \right) \leq n^{-d} + n^{-d'} + e^{-\frac{4t}{3n}}.$$

Proof. Assume that $I_{r,\Gamma}(t) = V$. Let $t_1, t_2, \dots, t_{n-1}$ be the integers such that $I_{r,\Gamma}(t_i) \neq I_{r,\Gamma}(t_i + 1)$ for each $i \in [1, n-1]$ and $0 \leq t_1 < t_2 < \dots < t_{n-1} < t$. Let $u_1, u_2, \dots, u_{n-1}$ be the agents where $t_i = t_{r,\Gamma}(u_i)$. Intuitively, u_i is the $(i+1)^{\text{st}}$ agent to be infected and t_i is its infection time in the epidemic starting from r under Γ. We let $u_0 = r$. Note that both u_i and t_i are random variables. Let Z_1 be the number of interactions that virtual agent $VA_{r,\gamma}(v)$ participates in among the $n-1$ interactions $\Gamma_{t_1}, \Gamma_{t_2}, \dots, \Gamma_{t_{n-1}}$. Let $Z_2 = VI_{r,\Gamma}(v,t) - Z_1$. As mentioned above, the parent-child relation based on the epidemic starting from r gives the infection path from r to v, denoted by $v_0 (= r), v_1, v_2, \dots, v_k = v$. (Note that k and each v_i are random variables.) For any j $(0 \leq j < t)$, let X_j be the indicator variable such that $X_j = 1$ holds if $VA_{r,\Gamma}(v)$ participates in Γ_t, otherwise $X_j = 0$. In other words, we define $X_j = 1 \Leftrightarrow (j \geq t_{r,\Gamma}(v) \wedge v \in \Gamma_j) \vee (\exists i : t_{r,\Gamma}(v_i) \leq j < t_{r,\Gamma}(v_{i+1}) \wedge v_i \in \Gamma_j)$. Let $X = \sum_{j \in \{t_1, t_2, \dots, t_{n-1}\} \setminus \{t_{r,\Gamma}(v_1), t_{r,\Gamma}(v_2), \dots, t_{r,\Gamma}(v_k)\}} X_j$. It trivially holds that $Z_1 = \sum_{j \in \{t_1, t_2, \dots, t_{n-1}\}} X_j = k + X$. Hence, $VI_{r,\Gamma}(v,t) = Z_1 + Z_2 = k + X + Z_2$. In the following, we give probabilistic upper bounds on k, X, and Z_2.

First, we focus on k, the length of the infection path from r to v. The index of v_i $(0 \leq i \leq k)$, denoted by x_i, is defined to be the integer such that $v_i = u_{x_i - 1}$, i.e., v_i is the x_i^{th} agent to be infected among the population. For example, $v_0 = r$ is the first infected agent and thus $x_0 = 1$. The parent of each u_j is chosen uniformly at random among $u_0, u_1, \dots, u_{j-1}$. Therefore, $\Pr(x_i \geq 2x_{i-1}) \geq 1/2$ for $i = 1, 2, \dots, k$. If $k \geq 4d\lceil \log_2 n \rceil$, then the event $x_i \geq 2x_{i-1}$ must not happen more than $\lceil \log_2 n \rceil$ times for $i = 1, 2, \dots, 4d\lceil \log_2 n \rceil$.

Hence, letting Y be a binomial random variable such that $Y \sim B(4d\lceil \log_2 n \rceil, 1/2)$, we have

$$\begin{aligned}
\Pr(k \geq 4d\lceil \log_2 n \rceil \mid I_{r,\Gamma}(t) = V) &\leq \Pr(Y \leq \lceil \log_2 n \rceil) \\
&= \Pr\left(Y \leq \frac{\mathbf{E}[Y]}{2d}\right) \\
&\leq \exp\left(-\frac{1}{2} \cdot \left(\frac{2d-1}{2d}\right)^2 \cdot 2d\lceil \log_2 n \rceil\right) \\
&\leq \exp\left(-d \ln n \cdot \log_2 e \cdot \left(\frac{2d-1}{2d}\right)^2\right) \\
&\leq n^{-d},
\end{aligned}$$

where we use (3) in Lemma 2 for the second inequality and $\forall d \geq 3 : \log_2 e \cdot ((2d-1)/2d)^2 \geq 1$ for the last inequality ($\log_2 e \cdot (5/6)^2 = 1.00187\dots$).

Next, we focus on the random number $X = \sum_{j \in \{t_1, t_2, \dots, t_{n-1}\} \setminus \{t_{r,\Gamma}(v_1), t_{r,\Gamma}(v_2), \dots, t_{r,\Gamma}(v_k)\}} X_j$. For any $j \in \{t_1, t_2, \dots, t_{n-1}\} \setminus \{t_{r,\Gamma}(v_1), t_{r,\Gamma}(v_2), \dots, t_{r,\Gamma}(v_k)\}$, we have $\Pr(X_j = 1) = 1/i$, where $j = t_i$. Therefore, letting $W_1, W_2, \dots, W_{n-1}$ be a sequence of independent Poisson trials with probability $\Pr(W_i = 1) = 1/i$, we have

$$\Pr(X \geq 2d' H_n \mid I_{r,\Gamma}(t) = V) \leq \Pr\left(\sum_{i=1}^{n-1} W_i \geq 2d' H_{n-1}\right) \leq 2^{-2d' H_{n-1}} \leq e^{-d' H_{n-1}} \leq n^{-d'},$$

where we use $d' \geq 3$, $H_{n-1} = \mathbf{E}[\sum_{i=1}^{n-1} W_i]$, (2) in Lemma 2 for the second inequality, and $H_{n-1} > \ln n$ for the last inequality.

Finally, we focus on Z_2, the number of non-infection interactions that virtual agent $VA_{r,\Gamma}(v)$ participates in among $\Gamma_0, \Gamma_1, \dots, \Gamma_{t-1}$. Under the condition that no agent is newly infected by an interaction Γ_j, the probability that $VA_{r,\Gamma}(v)$ participates in Γ_j is at most $4/n$ [14]. This is because, letting $m = |I_{r,\Gamma}(j)|$ and ${}_0C_2 = {}_1C_2 = 0$, the probability is exactly $\frac{m-1}{{}_mC_2 + {}_{n-m}C_2}$, which is at most $4/n$ regardless of j (See Appendix in [14]). Therefore, letting W' be a binomial random variable such that $W' \sim B(t, 4/n)$, we have

$$\Pr\left(Z_2 \geq \frac{4d''t}{n} \;\middle|\; I_{r,\Gamma}(t) = V\right) \leq \Pr\left(W' \geq \frac{4d''t}{n}\right) \leq 2^{-4d''t/n} \leq e^{-2d''t/n},$$

where we use $d'' \geq 6$ and (2) in Lemma 2 for the second inequality. Similarly, we have

$$\Pr\left(Z_2 \geq \frac{8t}{n} \;\middle|\; I_{r,\Gamma}(t) = V\right) \leq \Pr\left(W' \geq \frac{8t}{n}\right) \leq e^{-4t/3n},$$

where we use (1) in Lemma 2 for the second inequality.

The two inequalities of the lemma follow from the above probabilistic upper bounds on k, X, and Z_2. ◀

From Lemma 3, Lemma 4, and Lemma 5, we obtain Lemma 6 below.

▶ **Lemma 6.** *Let P be any protocol and* `var` *be a propagating variable of P. Let $C_0 \in \mathcal{C}_{\mathrm{all}}(P)$ and $\Xi_P(C_0, \Gamma) = C_0, C_1, \dots$. Then the following inequalities hold for any $r \in V$, $t \geq 1$,*

$d \geq 3$, $d' \geq 3$, *and* $d'' \geq 6$*:*

$$\Pr\left(\forall v \in V : C_{2t}(v).\mathtt{var} > C_0(r).\mathtt{var} - 4d\lceil \log_2 n\rceil - 2d'H_n - \frac{8d''t}{n}\right)$$
$$\geq 1 - n\left(n^{-d} + n^{-d'} + e^{-\frac{4d''t}{n}} + e^{-\frac{t}{n}}\right)$$
$$\Pr\left(\forall v \in V : C_{2t}(v).\mathtt{var} > C_0(r).\mathtt{var} - 4d\lceil \log_2 n\rceil - 2d'H_n - \frac{16t}{n}\right)$$
$$\geq 1 - n\left(n^{-d} + n^{-d'} + e^{-\frac{8t}{3n}} + e^{-\frac{t}{n}}\right).$$

Lemma 6 is formalized for general applications: the lemma can be used for any protocol and any of its propagating variables. In this paper, we use the following two corollaries.

▶ **Corollary 7.** *Let* $C_0 \in \mathcal{L}_n$ *and* $\Xi_{P_{\mathrm{PL}}}(C_0, \Gamma) = C_0, C_1, \ldots$ *. Then:*

$$\Pr\left(C_{6cn\lceil \ln N\rceil^2} \in \mathcal{G}_{\mathrm{half}}\right) \geq 1 - 3n^{-10c}.$$

Proof. There exists an agent $r \in V$ such that $C_0(r).\mathtt{leader}$ holds since $C_0 \in \mathcal{L}_n$. The value of the leader-timer in r may not be $t_{\max}$ in the initial configuration C_0. However, in the first interaction that r participates in, r resets the timers of both of the interacting agents to $t_{\max}$. Therefore, we can assume that $C_0(r).\mathtt{timer}_L = t_{\max}$. Assigning $t = 3cn\lceil \ln N\rceil^2$, $d = d' = 11c$, and $d'' = 6$ to the first inequality of Lemma 6 yields the result, because we have:

$$\begin{aligned}4d\lceil \log_2 n\rceil + 2d'H_n + \frac{8d''t}{n} &\leq 44c\lceil \log_2 n\rceil + 22c(1+\ln n) + 144c\lceil \ln N\rceil^2\\ &\leq 44c(1+\log_2 e\lceil \ln N\rceil) + 22c(1+\lceil \ln N\rceil) + 144c\lceil \ln N\rceil^2\\ &\leq 144c\lceil \ln N\rceil^2 + 86c\lceil \ln N\rceil + 66c\\ &\leq \frac{t_{\max}}{2},\end{aligned}$$

where we use $t_{\max} = 720c\lceil \ln N\rceil^2$ in the last inequality, and we have:

$$n\left(n^{-d} + n^{-d'} + e^{-\frac{4d''t}{n}} + e^{-\frac{t}{n}}\right) \leq n(2n^{-11c} + n^{-72c\lceil \ln N\rceil} + N^{-15c}) \leq 3n^{-10c}.$$ ◀

▶ **Corollary 8.** *Let* $C_0 \in \mathcal{C}_{\mathrm{all}}(P_{\mathrm{PL}})$ *and* $\Xi_{P_{\mathrm{PL}}}(C_0, \Gamma) = C_0, C_1, \ldots$ *. Assume* $C_0(r).\mathtt{virus} = t_{\mathrm{virus}}$ *for some* $r \in V$*. Then:*

$$\Pr\left(\forall v \in V : C_{4n\lceil \ln N\rceil}(v).\mathtt{virus} > 0\right) \geq 1 - 2/n.$$

Proof. Assigning $t = 2n\lceil \ln N\rceil$ and $d = d' = 3$ to the second inequality of Lemma 6 we have

$$\begin{aligned}4d\lceil \log_2 n\rceil + 2d'H_n + \frac{16t}{n} &\leq 12\lceil \log_2 n\rceil + 6(1+\lceil \ln N\rceil) + 32\lceil \ln N\rceil\\ &\leq 12(1+\log_2 e\lceil \ln N\rceil) + 6(1+\lceil \ln N\rceil) + 32\lceil \ln N\rceil\\ &\leq 56\lceil \ln N\rceil + 18\\ &< t_{\mathrm{virus}},\end{aligned}$$

where we use the assumption $\lceil \ln N\rceil \geq 5$ in the last inequality, and we have

$$n\left(n^{-d} + n^{-d'} + e^{-\frac{8t}{3n}} + e^{-\frac{t}{n}}\right) \leq n(2n^{-3} + N^{-5} + N^{-2}) \leq 2/n,$$

where we use the assumption $n \geq 55$ in the last inequality. ◀

4.2 Expected Holding Time

We prove (5) in this section, that is, $\min_{C \in \mathcal{S}} \mathrm{EIH}_{P_{\mathrm{PL}}}(C, LE) = \Omega\left(cn^{10c+1}\right)$. Let t be a positive integer and $\gamma = \gamma_0, \gamma_1, \dots$ be a schedule. Let $C_0 \in \mathcal{C}_{\mathrm{all}}(P_{\mathrm{PL}})$ and $\Xi_{P_{\mathrm{PL}}}(C_0, \gamma) = C_0, C_1, \dots$. We define the indicator variable: $L^+_{C_0,\gamma}(t) = \top$ if a timeout occurs and a new leader is created in the prefix $C_0, C_1, \dots, C_t$ of length t+1 of the execution that is, at least one of the interactions $\gamma(0), \gamma(1), \dots, \gamma(t-1)$ causes a timeout in $\Xi_{P_{\mathrm{PL}}}(C_0, \gamma)$; Otherwise we let $L^+_{C_0,\gamma}(t) = \bot$. For convenience, we define $L^+_{C_0,\gamma}(0) = \bot$. Similarly, we define the indicator variable $L^0_{C_0,\gamma}(t)$ as follows: $L^0_{C_0,\gamma}(t) = \top$ if and only if the prefix of length $t+1$ of the execution includes a configuration where no leader exists. We also define $\tau = 144cn\lceil \ln N \rceil^2 = nt_{\mathrm{emit}}/5$. In the rest of this section, we prove that for any configuration $C_0 \in \mathcal{S}$:

$$\Pr\left(\neg L^+_{C_0,\Gamma}(\tau) \wedge \neg L^0_{C_0,\Gamma}(\tau) \wedge C_\tau \in \mathcal{S}\right) \geq 1 - O(n^{-10c}), \tag{6}$$

where $\Xi_{P_{\mathrm{PL}}}(C_0, \Gamma) = C_0, C_1, \dots$. From this inequality, we have $\min_{C \in \mathcal{S}} \mathrm{EIH}_{P_{\mathrm{PL}}}(C, LE) \geq (1 - O(n^{-10c})) \cdot (\tau + \min_{C \in \mathcal{S}} \mathrm{EIH}_{P_{\mathrm{PL}}}(C, LE))$, which yields (5), i.e., $\min_{C \in \mathcal{S}} \mathrm{EIH}_{P_{\mathrm{PL}}}(C, LE) = \Omega(cn^{10c+1})$.

Let v be an agent, $t \geq 1$ an integer, and $\gamma = \gamma_0, \gamma_1, \dots$ a schedule. We denote by $RI_\gamma(v, t)$ the number of interactions in which v participates among the first t interactions of γ (i.e., $\gamma_0, \gamma_1, \dots, \gamma_{t-1}$). We also define (asynchronous) *round time*. The first round time $\mathrm{RT}_\gamma(1)$ is the minimum t satisfying $\forall v \in V, \exists j \in [0, t-1] : v \in \gamma_j$. For any $i \geq 2$, we define the ith round time $\mathrm{RT}_\gamma(i)$ as the minimum t satisfying $\forall v \in V, \exists j \in [\mathrm{RT}_\gamma(i-1), t-1] : v \in \gamma_j$. For completeness, we define $\mathrm{RT}_\gamma(0) = 0$. Note that, for any $i \geq 1$, every agent $v \in V$ participates in at least one interaction in $\gamma_{\mathrm{RT}_\gamma(i-1)}, \dots, \gamma_{\mathrm{RT}_\gamma(i)-1}$.

▶ **Lemma 9.** $\Pr(\max_{v \in V} RI_\Gamma(v, \tau) \geq t_{\mathrm{emit}}/2) < N^{-30c+1}$.

Proof. Since each agent v participates in Γ_t with probability $2/n$ for any $t \geq 0$, we have $\mathbf{E}[RI_\Gamma(v, \tau)] = 2\tau/n = \frac{2}{5}t_{\mathrm{emit}}$. Therefore, $\Pr(RI_\Gamma(v, \tau) \geq \frac{1}{2}t_{\mathrm{emit}}) = \Pr(RI_\Gamma(v, \tau) \geq \frac{5}{4}\mathbf{E}[RI_\Gamma(v, \tau)]) \leq \exp(-\mathbf{E}[RI_\Gamma(v, \tau)]/48) = \exp(-6c\lceil \ln N \rceil^2) \leq N^{-30c}$ by the Chernoff bound (1) in Lemma 2 with $\delta = 1/4$, and by the assumption $\lceil \ln N \rceil \geq 5$. Hence, the lemma holds by the union bounds. ◀

To prove (6), we first give lower bounds on the probabilities of $\neg L^+_{C_0,\Gamma}(\tau)$ and $\neg L^0_{C_0,\Gamma}(\tau)$ in Lemmas 10 and 11, respectively. Then, we give lower bounds on the probability of $C_\tau \in \mathcal{S}$ by Lemmas 14 and 15.

▶ **Lemma 10.** *Let $C_0 \in \mathcal{G}_{\mathrm{half}}$. Then,* $\Pr(L^+_{C_0,\Gamma}(\tau)) \leq N^{-30c+1}$.

Proof. Since $t_{\max} = t_{\mathrm{emit}}$, and the leader-timer of every agent is no less than $t_{\max}/2$ in $C_0 \in \mathcal{G}_{\mathrm{half}}$, a timeout happens within the first τ interactions with probability at most N^{-30c+1}, by Lemma 9. ◀

▶ **Lemma 11.** *Let $C_0 \in \mathcal{L}_n \cap (\mathcal{L}_{\mathrm{safe}} \cup \mathcal{V}_{\mathrm{clean}})$. Then,* $\Pr(L^0_{C_0,\Gamma}(\tau)) \leq N^{-30c+1}$.

Proof. First, consider the case $C_0 \in \mathcal{L}_{\mathrm{safe}}$. The population has a shielded leader v_l whose individual timer is at least $t_{\mathrm{emit}}/2$ at C_0. Therefore, $L^0_{C_0,\Gamma}(\tau)$ holds only if v_l participates in $t_{\mathrm{emit}}/2$ or more interactions and becomes unshielded within the first τ interactions. Next, consider the case $C_0 \in \mathcal{L}_n \cap \mathcal{V}_{\mathrm{clean}}$. One or more leaders exist, but no virus exists in the population in C_0. The leaders are never killed (become non-leaders) until viruses appear in the population. Therefore, $L^0_{C_0,\Gamma}(\tau)$ holds only if some leader, say v_l, creates a new virus.

However, v_l resets its individual timer to t_{emit} and becomes shielded when it creates a new virus. Therefore, $L^0_{C_0,\Gamma}(\tau)$ holds only if v_l participates in t_{emit} or more interactions among the first τ interactions. Thus, in both cases, $L^0_{C_0,\Gamma}(\tau)$ holds only if some agent participates in $t_{\mathrm{emit}}/2$ or more interactions among the first τ interactions. By Lemma 9, this necessary condition holds with probability at most N^{-30c+1}, which yields the lemma. ◀

▶ **Lemma 12.** $\Pr(\mathrm{RT}_\Gamma(i) \ge 2in(1+\lceil \ln n \rceil)) \le ne^{-i/4}$ *for any* $i \ge 1$.

Proof. Each round finishes when every agent $v \in V$ has interacted during that round. Consider the case that s ($s \ge 1$) agents have not yet interacted in round j. One of these s agents participates in the next interaction with probability $\frac{{}_sC_2 + s(n-s)}{{}_nC_2} \ge \frac{s}{n}$. Let $X_{1,s}, X_{2,s}, \dots, X_{i,s}$ be independent random variables each of which corresponds to the number of trials needed to reach the first success where the success probability of each trial is s/n. We have

$$\begin{aligned}\Pr(\mathrm{RT}_\Gamma(i) \ge 2in(1+\lceil \ln n \rceil) &\le \Pr\left(\sum_{j=1}^{i}\sum_{s=1}^{n} X_{j,s} \ge 2in(1+\lceil \ln n \rceil)\right) \\ &\le \Pr\left(\sum_{s=1}^{n}\sum_{j=1}^{i} X_{j,s} \ge 2inH_n\right) \\ &\le \Pr\left(\sum_{s=1}^{n}\sum_{j=1}^{i} X_{j,s} \ge \sum_{s=1}^{n}\frac{2in}{s}\right) \le \sum_{s=1}^{n}\Pr\left(\sum_{j=1}^{i} X_{j,s} \ge \frac{2in}{s}\right).\end{aligned}$$

For a binomial random variable $Y_s \sim B(\lceil \frac{2in}{s} \rceil, \frac{s}{n})$, we have $\Pr(\sum_{j=1}^{i} X_{j,s} \ge \frac{2in}{s}) = \Pr(\sum_{j=1}^{i} X_{j,s} \ge \lceil \frac{2in}{s} \rceil) \le \Pr(Y_s \le i)$. Hence

$$\Pr\left(\sum_{j=1}^{i} X_{j,s} \ge \frac{2in}{s}\right) \le \Pr(Y_s \le i) \le \Pr\left(Y_s \le \frac{1}{2}\cdot \mathbf{E}[Y_s]\right) \le e^{-\mathbf{E}[Y_s]/8} \le e^{-i/4},$$

where we use Chernoff bound given as (3) in Lemma 2 for the third inequality. ◀

▶ **Corollary 13.** $\Pr(\mathrm{RT}_\Gamma(t_{\mathrm{virus}}) \ge \tau) \le N^{-15c+1}$.

Proof. Since we assume $\lceil \ln N \rceil \ge 5$, Lemma 12 yields $\Pr(\mathrm{RT}_\Gamma(t_{\mathrm{virus}}) \ge \tau) = \Pr(\mathrm{RT}_\Gamma(t_{\mathrm{virus}}) \ge 144cn\lceil \ln N \rceil^2) \le \Pr(\mathrm{RT}_\Gamma(t_{\mathrm{virus}}) \ge 120cn\lceil \ln N \rceil(1+\lceil \ln n \rceil)) \le \Pr(\mathrm{RT}_\Gamma(c \cdot t_{\mathrm{virus}}) \ge 2(c \cdot t_{\mathrm{virus}})n(1+\lceil \ln n \rceil)) \le ne^{-c\cdot t_{\mathrm{virus}}/4} \le N^{-15c+1}$. ◀

▶ **Lemma 14.** *Let* $C_0 \in \mathcal{C}_{\mathrm{all}}(P_{\mathrm{PL}})$ *and* $\Xi_{P_{\mathrm{PL}}}(C_0, \Gamma) = C_0, C_1, \dots$. *Then,* $\Pr(C_\tau \notin \mathcal{L}_{\mathrm{safe}} \cup \mathcal{V}_{\mathrm{clean}}) \le 2N^{-15c+1}$.

Proof. Consider the case that every agent has fewer than $t_{\mathrm{emit}}/2$ interactions during the first τ interactions, and the $t_{\mathrm{virus}}^{\mathrm{th}}$ round finishes during the first τ interactions. Thanks to the latter condition, viruses disappear from the population and no virus exists in C_τ, i.e., $C_\tau \in \mathcal{V}_{\mathrm{clean}}$, if no leader creates a new virus within the first τ interactions. If some leader $v_l \in V$ creates a new virus during that period, then $v_l.\mathtt{leader} \wedge v_l.\mathtt{shield} \wedge v_l.\mathtt{timer}_I \ge t_{\mathrm{emit}}/2$ in C_τ, thanks to the former condition, which implies $C_\tau \in \mathcal{L}_{\mathrm{safe}}$. This is because v_l resets its individual timer to t_{emit} and becomes shielded at the time it creates a new virus. The probability that both conditions hold is at least $1 - N^{-30c+1} + N^{-15c+1} > 1 - 2N^{-15c+1}$ by Lemma 9 and Corollary 13. ◀

▶ **Lemma 15.** *Let* $C_0 \in \mathcal{C}_{\mathrm{all}}(P_{\mathrm{PL}})$ *and* $\Xi_{P_{\mathrm{PL}}}(C_0,\gamma) = C_0, C_1, \dots$ *. Then* $\Pr(C_\tau \notin \mathcal{G}_{\mathrm{half}} \mid C_{\tau-6cn\lceil \ln N\rceil^2} \in \mathcal{L}_n) \le 3n^{-10c}$.

Proof. Immediate from Corollary 7. ◀

The inequality (6) follows from Lemma 10, Lemma 11, Lemma 14, and Lemma 15. Thus, we obtain the following lemma from the discussion in the beginning of this sub-section.

▶ **Lemma 16.** $\min_{C\in\mathcal{S}} \mathrm{EIH}_{P_{\mathrm{PL}}}(C, LE) = \Omega\left(cn^{10c+1}\right)$.

4.3 Expected Convergence Time

We prove (4) in this section, that is, $\max_{C\in\mathcal{C}_{\mathrm{all}}(P_{\mathrm{PL}})} \mathrm{EIC}_{P_{\mathrm{PL}}}(C,\mathcal{S}) = O(cn\log n\cdot\log^2 N)$. Recall that $\mathcal{S} = \mathcal{L}_1 \cap \mathcal{G}_{\mathrm{half}} \cap (\mathcal{L}_{\mathrm{safe}} \cup \mathcal{V}_{\mathrm{clean}})$. We make use of the fact that $\tau = 144cn\lceil \ln N\rceil^2$.

▶ **Lemma 17.** *Let* $C_0 \in \mathcal{L}_{\mathrm{safe}} \cup \mathcal{V}_{\mathrm{clean}}$ *and* $\Xi_{P_{\mathrm{PL}}}(C_0,\Gamma) = C_0, C_1, \dots$ *. Then* $\Pr(\exists j \in [0, 12\tau\lceil \ln N\rceil] : C_j \in \mathcal{L}_n \cap (\mathcal{L}_{\mathrm{safe}} \cup \mathcal{V}_{\mathrm{clean}})) \ge 1 - O(N^{-1})$.

Proof. Since $C_0 \in \mathcal{L}_{\mathrm{safe}} \cup \mathcal{V}_{\mathrm{clean}}$, we have $C_0 \in \mathcal{L}_n \cap (\mathcal{L}_{\mathrm{safe}} \cup \mathcal{V}_{\mathrm{clean}})$ if $C_0 \in \mathcal{L}_n$. Hence, it suffices to consider only the case that $C_0 \notin \mathcal{L}_n$. In this case, $C_0 \in \mathcal{V}_{\mathrm{clean}}$ also holds because $C_0 \in \mathcal{L}_{\mathrm{safe}} \cup \mathcal{V}_{\mathrm{clean}}$ and $\mathcal{L}_{\mathrm{safe}} \subset \mathcal{L}_n$. To conclude, we need only consider the case that there exists no virus and no leader in the population at C_0. Staring from such a configuration, the population reaches a configuration in $\mathcal{L}_n \cap (\mathcal{L}_{\mathrm{safe}} \cup \mathcal{V}_{\mathrm{clean}})$ immediately after a new leader is created by the timeout. While no leader exists in the population, $\max_{v\in V} v.\mathtt{timer}_L$ is monotonically non-increasing and decreases at least by one during each asynchronous round. Therefore, the timeout occurs within $t_{\max}$ rounds. Lemma 12 guarantees that $\Pr(\mathrm{RT}_\Gamma(t_{\max}) \ge 2nt_{\max}(1+\lceil \ln n\rceil) \le ne^{-t_{\max}/4} = O(N^{-1})$. Since $\lceil \ln N\rceil \ge 5$, we have $2nt_{\max}(1+\lceil \ln n\rceil) = 10\tau(1+\lceil \ln n\rceil) \le 12\tau\lceil \ln N\rceil$, which yields the lemma. ◀

▶ **Lemma 18.** *Let* $C_0 \in \mathcal{L}_n \cap (\mathcal{L}_{\mathrm{safe}} \cup \mathcal{V}_{\mathrm{clean}})$ *and* $\Xi_{P_{\mathrm{PL}}}(C_0,\Gamma) = C_0, C_1, \dots$ *. Then* $\Pr(C_\tau \in \mathcal{L}_n \cap \mathcal{G}_{\mathrm{half}} \cap (\mathcal{L}_{\mathrm{safe}} \cup \mathcal{V}_{\mathrm{clean}})) \ge 1 - O(N^{-1})$.

Proof. Immediate from Lemma 11, Lemma 14, and Lemma 15. ◀

▶ **Lemma 19.** $\Pr(\min_{v\in V} RI_\Gamma(v, 5\tau) \le t_{\mathrm{emit}}) < N^{-900c+1}$.

Proof. Each agent participates in Γ_t with probability $\frac{2}{n}$, for any $t \ge 0$. Therefore, the Chernoff bound given as (3) in Lemma 2 with $\delta = 1/2$ yields $\Pr(RI_\Gamma(v, 5\tau) \le t_{\mathrm{emit}}) \le e^{t_{\mathrm{emit}}/4} < N^{-900c}$. The lemma holds by the union bounds. ◀

Intuitively, the following lemma guarantees that the number of leaders decreases at least by half during every 15τ interactions with probability close to $1/4$, and never increases with probability $1 - O(N^{-1})$ after the population enters a configuration in $C_0 \in \mathcal{L}_n \cap \mathcal{G}_{\mathrm{half}} \cap (\mathcal{L}_{\mathrm{safe}} \cup \mathcal{V}_{\mathrm{clean}})$.

▶ **Lemma 20.** *Let* $i \in [0, n-1]$. *Let* $C_0 \in \mathcal{L}_{1+i} \cap \mathcal{G}_{\mathrm{half}} \cap (\mathcal{L}_{\mathrm{safe}} \cup \mathcal{V}_{\mathrm{clean}})$ *and* $\Xi_{P_{\mathrm{PL}}}(C_0,\Gamma) = C_0, C_1, \dots$ *. The following inequalities hold:*

$$\Pr(C_{15\tau} \in \mathcal{L}_{1+\lfloor i/2\rfloor}) \ge 1/4 - O(n^{-1}) \tag{7}$$

$$\Pr(C_{15\tau} \in \mathcal{L}_{1+i} \cap \mathcal{G}_{\mathrm{half}} \cap (\mathcal{L}_{\mathrm{safe}} \cup \mathcal{V}_{\mathrm{clean}})) \ge 1 - O(N^{-1}). \tag{8}$$

Proof. In this proof, we use notation "with high probability" to represent "with probability $1-O(n^{-1})$". By repeated application of Lemma 10, Lemma 11, Lemma 14, and Lemma 15, it holds with high probability that no new leader is created and at least one leader always exists in the population in $C_0, C_1, \dots, C_{15\tau}$. Moreover, Lemma 19 guarantees that the individual timer of every agent reaches zero in every 5τ interactions with high probability. In particular, for all $v \in V$, $v.\mathtt{timer}_I$ reaches zero at least once in the middle period $C_{5\tau}, C_{5\tau+1}, \dots, C_{10\tau}$ with high probability. Let V_L be the set of all leaders in C_0 and $i' = |V_L| - 1$. Note that $0 \le i' \le i$ since $C_0 \in \mathcal{L}_{1+i}$. Since $\neg L^0_{C_0,\Gamma}(15\tau)$ with high probability, some $v_l \in V_L$ creates a new virus with probability at least $1/2 - O(n^{-1})$. This is because, when the individual timer of a leader reaches zero in an interaction, that leader creates a new virus with probability $1/2$ (i.e., if it is the initiator of the current interaction). The virus propagates to the whole population within $4n \log n$ ($< 5\tau$) interactions thereafter with high probability by Corollary 8. At this time, with probability at least $1/2$, independently of configuration C_0, no less than $\lceil i'/2 \rceil$ agents in $V_L \setminus \{v_l\}$ are unshielded. This is because the individual timer of every $v \in V_L \setminus \{v_l\}$ reaches zero before the time, and then becomes unshielded with probability $1/2$. We make the margin period $C_0, C_1, \dots, C_{5\tau}$ for this reason. To conclude, no fewer than $\lceil i'/2 \rceil$ leaders in $V_L \setminus \{v_l\}$ are killed and at most $1 + \lfloor i'/2 \rfloor \le 1 + \lfloor i/2 \rfloor$ leaders survive (i.e., remain leaders) in $C_{15\tau}$, with probability $1/4 - O(n^{-1})$. Therefore, we obtain (7) since $\neg L^+_{C_0,\Gamma}(15\tau) \wedge \neg L^0_{C_0,\Gamma}(15\tau)$ holds with high probability. Inequality (8) is guaranteed simply by applying Lemma 10, Lemma 11, Lemma 14, and Lemma 15 repeatedly (fifteen times). ◀

The following corollary follows from Lemma 20.

▶ **Corollary 21.** *Let* $C_0 \in \mathcal{L}_n \cap \mathcal{G}_{\mathrm{half}} \cap (\mathcal{L}_{\mathrm{safe}} \cup \mathcal{V}_{\mathrm{clean}})$ *and* $\Xi_{P_{\mathrm{PL}}}(C_0, \Gamma) = C_0, C_1, \dots$. *Then, there exists some integer* $w = O(\tau \log n)$ *such that* $\Pr(C_w \in \mathcal{S}) \ge 1 - O\left(\frac{\log n}{N}\right)$.

▶ **Lemma 22.** $\max_{C \in \mathcal{C}_{\mathrm{all}}(P_{\mathrm{PL}})} \mathrm{EIC}_{P_{\mathrm{PL}}}(C, \mathcal{S}) = O\left(cn \log n \cdot \log^2 N\right)$ *holds.*

Proof. By Lemma 14, Lemma 17, Lemma 18, and Corollary 21, we have the following inequality:

$$\max_{C \in \mathcal{C}_{\mathrm{all}}(P_{\mathrm{PL}})} \mathrm{EIC}_{P_{\mathrm{PL}}}(C, \mathcal{S}) \le O(\tau \log n) + O\left(\frac{\log n}{N}\right) \cdot \max_{C \in \mathcal{C}_{\mathrm{all}}(P_{\mathrm{PL}})} \mathrm{EIC}_{P_{\mathrm{PL}}}(C, \mathcal{S}).$$

Solving this inequality yields $\max_{C \in \mathcal{C}_{\mathrm{all}} P_{\mathrm{PL}}} \mathrm{EIC}_{P_{\mathrm{PL}}}(C, \mathcal{S}) = O(\tau \log n) = O(cn \log n \cdot \log^2 N)$. ◀

▶ **Theorem 23.** *Protocol* P_{PL} *is an* $(O(cn \log n \cdot \log^2 N), \Omega(cn^{10c+1}))$*- loosely-stabilizing leader election protocol.*

Thus, in terms of parallel time, P_{PL} is a loosely-stabilizing leader election algorithm with polylogarithmic convergence time ($O(c \log n \cdot \log^2 N) \subseteq O(c \log^3 N)$) and arbitrarily large polynomial holding time ($\Omega(cn^{10c})$).

5 Conclusion

We have presented a loosely-stabilizing leader election protocol with polylogarithmic convergence time. Given an upper bound N of n and a parameter c, our protocol elects a unique leader in the population within $O(c \log^3 N)$ parallel time starting from any configuration, and keeps the unique leader for $\Omega(cn^{10c})$ parallel time.

References

1. Dan Alistarh and Rati Gelashvili. Polylogarithmic-time leader election in population protocols. In *Proceedings of the 42nd International Colloquium on Automata, Languages, and Programming*, pages 479–491. Springer, 2015.
2. Dan Alistarh, Rati Gelashvili, and Milan Vojnović. Fast and exact majority in population protocols. In *the 34th ACM Symposium on Principles of Distributed Computing*, pages 47–56, 2015.
3. D. Angluin, J Aspnes, Z. Diamadi, M.J. Fischer, and R. Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, 18(4):235–253, 2006.
4. D. Angluin, J. Aspnes, M. J Fischer, and H. Jiang. Self-stabilizing population protocols. *ACM Transactions on Autonomous and Adaptive Systems*, 3(4):13, 2008.
5. Dana Angluin, James Aspnes, and David Eisenstat. Fast computation by population protocols with a leader. *Distributed Computing*, 21(3):183–199, 2008.
6. J. Beauquier, P. Blanchard, and J. Burman. Self-stabilizing leader election in population protocols over arbitrary communication graphs. In *International Conference on Principles of Distributed Systems*, pages 38–52, 2013.
7. S. Cai, T. Izumi, and K. Wada. How to prove impossibility under global fairness: On space complexity of self-stabilizing leader election on a population protocol model. *Theory of Computing Systems*, 50(3):433–445, 2012.
8. D. Canepa and M. G. Potop-Butucaru. Stabilizing leader election in population protocols, 2007. http://hal.inria.fr/inria-00166632.
9. M. J. Fischer and H. Jiang. Self-stabilizing Leader Election in Networks of Finite-State Anonymous Agents. In *International Conference on Principles of Distributed Systems*, pages 395–409, 2006. doi:10.1007/11945529_28.
10. Leszek Gąsieniec and Grzegorz Staehowiak. Fast space optimal leader election in population protocols. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2653–2667. SIAM, 2018.
11. T. Izumi. On Space and Time Complexity of Loosely-Stabilizing Leader Election. In *International Colloquium on Structural Information and Communication Complexity*, pages 299–312, 2015.
12. M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.
13. R. Mizoguchi, H. Ono, S. Kijima, and M. Yamashita. On space complexity of self-stabilizing leader election in mediated population protocol. *Distributed Computing*, 25(6):451–460, 2012.
14. Y. Sudo, J. Nakamura, Y. Yamauchi, F. Ooshita, H. Kakugawa, and T. Masuzawa. Loosely-stabilizing leader election in a population protocol model. *Theoretical Computer Science*, 444:100–112, 2012.
15. Y. Sudo, F. Ooshita, H. Kakugawa, and T. Masuzawa. Loosely-Stabilizing Leader Election on Arbitrary Graphs in Population Protocols. In *International Conference on Principles of Distributed Systems*, pages 339–354, 2014.
16. Y. Sudo, F. Ooshita, H. Kakugawa, and T. Masuzawa. Loosely-stabilizing Leader Election on Arbitrary Graphs in Population Protocols without Identifiers nor Random Numbers. In *International Conference on Principles of Distributed Systems*, 2015.
17. Yuichi Sudo, Toshimitsu Masuzawa, Ajoy K Datta, and Lawrence L Larmore. The Same Speed Timer in Population Protocols. In *the 36th IEEE International Conference on Distributed Computing Systems*, pages 252–261, 2016.
18. X. Xu, Y. Yamauchi, S. Kijima, and M. Yamashita. Space Complexity of Self-Stabilizing Leader Election in Population Protocol Based on k-Interaction. In *Symposium on Self-Stabilizing Systems*, pages 86–97, 2013.

Self-Stabilizing Token Distribution with Constant-Space for Trees

Yuichi Sudo
Graduate School of Information Science and Technology, Osaka University, Japan
y-sudou@ist.osaka-u.ac.jp

Ajoy K. Datta
Department of Computer Science, University of Nevada, Las Vegas, USA
ajoy.datta@unlv.edu

Lawrence L. Larmore
Department of Computer Science, University of Nevada, Las Vegas, USA
lawrence.larmore@unlv.edu

Toshimitsu Masuzawa
Graduate School of Information Science and Technology, Osaka University, Japan
masuzawa@ist.osaka-u.ac.jp

Abstract

Self-stabilizing and silent distributed algorithms for token distribution in rooted tree networks are given. Initially, each process of a graph holds at most ℓ tokens. Our goal is to distribute the tokens in the whole network so that every process holds exactly k tokens. In the initial configuration, the total number of tokens in the network may not be equal to nk where n is the number of processes in the network. The root process is given the ability to create a new token or remove a token from the network. We aim to minimize the convergence time, the number of token moves, and the space complexity. A self-stabilizing token distribution algorithm that converges within $O(n\ell)$ asynchronous rounds and needs $\Theta(nh\epsilon)$ redundant (or unnecessary) token moves is given, where $\epsilon = \min(k, \ell - k)$ and h is the height of the tree network. Two novel ideas to reduce the number of redundant token moves are presented. One reduces the number of redundant token moves to $O(nh)$ without any additional costs while the other reduces the number of redundant token moves to $O(n)$, but increases the convergence time to $O(nh\ell)$. All algorithms given have constant memory at each process and each link register.

2012 ACM Subject Classification Theory of computation → Self-organization

Keywords and phrases token distribution, self-stabilization, constant-space algorithm

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.31

Related Version The brief announcement version of this paper is published in [16].

Acknowledgements This work was supported by JSPS KAKENHI Grant Numbers 17K19977 and 18K18000, and Japan Science and Technology Agency(JST) SICORP.

1 Introduction

The token distribution problem was originally defined by Peleg and Upfal in their seminal paper [15]. Consider a network of n processes and n tokens. Initially, the tokens are arbitrarily distributed among processes but with up to a maximum of ℓ tokens in any process. The problem is to distribute the tokens among the processes such that every process ends up with exactly one token. The above problem was redefined in another paper by the same

© Yuichi Sudo, Ajoy K. Datta, Lawrence L. Larmore, and Toshimitsu Masuzawa;
licensed under Creative Commons License CC-BY
22nd International Conference on Principles of Distributed Systems (OPODIS 2018).
Editors: Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira; Article No. 31; pp. 31:1–31:16

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

authors [14] by considering m tokens instead of n tokens. The goal in this case is to reach a configuration at which there are either $\lfloor m/n \rfloor$ or $\lceil m/n \rceil$ tokens at each process. The token distribution problem is a form of the load balancing problem in distributed systems. A token can be considered to represent a unit of task (or load) of a process. The solution to the token distribution problem provides a solution of the load balancing problem, where the goal is to maintain the loads of process as evenly as possible.

The fault-tolerant (self-stabilizing) version of the problem of load balancing in distributed systems was first considered by Arora and Gouda [1]. A self-stabilizing algorithm has two properties, convergence and stability. The convergence property states that regardless of the initial number of tokens in the processes, when the faults cease to occur, that is, the environment no longer changes the load, the execution will reach in finite steps a state where the loads of processes are balanced. The stability property avoids the possibility of any execution shifting any unit of load between any two processes forever.

In [1] and many other papers on token distribution papers, a special property (referred to as *constraint* in [1]) is maintained. This property specifies that during any execution of the load balancing or token distribution algorithm, no new tokens are produced and no tokens are consumed by any process. However, that constraint cannot be maintained in our work due to the nature of the problem. The token distribution problem solved in this paper requires storing a fixed number (k) of tokens at each process. As we deal with self-stabilizing systems, the network (tree in this paper) can start in an arbitrary configuration where the total number of tokens in the network may not be exactly equal to nk. Instead, each process holds an arbitrary number, from zero to ℓ, of tokens in an initial configuration. Thus, our algorithm must make an exception to the constraint above. We assume that only the root process can push/pull tokens to/from the external store as needed.

We present three silent and self-stabilizing token distribution algorithms for rooted tree networks in this paper. The performances of the algorithms are summarized in Table 1. First, we present a self-stabilizing token distribution algorithm *Base*. This algorithm has the optimal convergence time, $O(n\ell)$ (asynchronous) rounds. (As we will see in Section 3, any self-stabilizing token distribution algorithm requires convergence time of $\Omega(n\ell)$ rounds.) However, *Base* may have a large number of redundant token moves; $\Theta(nh\epsilon)$ *redundant* (or unnecessary) token moves occur in the worst case where $\epsilon = \min(k, \ell - k)$, where h is the height of the tree network. Next, we combine the algorithm *Base* with a synchronizer or with PIF waves to reduce redundant token moves, which results in *SyncTokenDist* or *PIFTokenDist*, respectively. Algorithm *SyncTokenDist* reduces the number of redundant token moves to $O(nh)$ without additional costs, while *PIFTokenDist* drastically reduces the number of redundant token moves to the asymptotically optimal value, i.e., $O(n)$, at the expense of increasing convergence time from $O(n\ell)$ to $O(nh\ell)$ rounds. The work space space, i.e., the amount of memory needed to store information except for tokens, of all our algorithms are constant both per process and per link register.

1.1 Related Work

The token distribution problem is introduced by Peleg and Upfal in [15]. Another solution to the same problem is given by the same authors in [14]. In these papers, the problem is defined for general bounded degree graphs. Herley [8] gives another scheme to solve the problem and claims that his solution is more efficient if the time for the local computation steps is not ignored. Another version of the problem, called near-perfect token distribution problem is introduced in [2]. In this problem, at the termination of the algorithm, no more than $O(1)$ tokens can be present at any process. There is a variation of the near-perfect

Table 1 Token distribution algorithms for rooted trees. ($\epsilon = \min(k, \ell - k)$).

	Conv. Time	#Red. Token Moves	Work Space (Process,Link)
Base	$O(n\ell)$ rounds	$\Theta(nh\epsilon)$	$(0, O(1))$
SyncTokenDist	$O(n\ell)$ rounds	$O(nh)$	$(O(1), O(1))$
PIFTokenDist	$O(nh\ell)$ rounds	$O(n)$	$(O(1), O(1))$
LowerBounds	$\Omega(n\ell)$ rounds	$\Omega(n)$	-

token distribution problem, where the maximum difference of tokens between any pair of processes at the termination of the algorithm is non-constant. That is, the difference depends on some network parameter. Algorithms in [7, 10, 11] are in this category. Algorithms for general networks are given in [7], for complete binary trees in [10], and for meshes and torus networks in [11].

The token distribution problem for tree networks is stated, without any solution, by Peleg in [13]. In this paper, we solve the problem given in that paper. Another version of the token distribution problem on trees is stated in [12], where the total number of tokens, denoted by T, and the number of processes of the tree are computed by the algorithm. The tokens are then distributed perfectly among the processes in $O(Td)$ time, where d is the diameter of the tree. Although the asynchronous message passing model was used in their work, they used stronger communication primitives than the typical send/receive primitives. The solution in [9] is for tree networks, where processes use the knowledge of n. The *dimension-exchange* model used in [9] is also different from models used by the other algorithms discussed above.

The token distribution problem is a version of the load balancing problem. Arora and Gouda [1] give self-stabilizing load balancing algorithms for ring and tree networks. Starting from an arbitrary initial assignment of load (or tasks), their algorithms are guaranteed to converge to a state where the loads of any pair of processes differ by at most 1.

2 Preliminaries

2.1 Model of Computation

We consider a tree network $T = (V, E)$ where V is the set of n processes and E is the set of $n-1$ links. Let v_{root} be the root process of T, and let $p(v)$ be the parent of any process $v \neq v_{\text{root}}$. Let $N(v)$ be the neighbors of a process v. Let $C(v)$ be the set of children of v, i.e., all neighbors of v except its parent. Let T_v be sub-tree of T consisting of v and its descendants. Let $n_v = |T_v|$. The height h_v of a process v is the length of the longest path through T between v and a leaf of T_v. Let $h = h_{v_{\text{root}}}$.

Each link $\{u, v\} \in E$ has two *link registers* (or just *registers*) $r_{u,v}$ and $r_{v,u}$. We call $r_{u,v}$ an *output register* of u and an *input register* of v. A process can read its input and its output registers. but can write only to its output registers. Thus, neighboring processes u and v can communicate with each other through $r_{u,v}$ and $r_{v,u}$.

A process v holds at most ℓ tokens, each of which is a bit sequence of length b. These tokens are stored in a dedicated memory space of the process, called the *internal token store*, written v.`tokenStore`, We use a link register to send and receive a token between processes. Each register $r_{u,v}$ contains at most one token in a dedicated variable $r_{u,v}$.`token`. The root process v_{root} can also access the *external token store*, which contains infinitely many tokens. As we will see later, v_{root} can reduce the total number of tokens in the tree, i.e., the tokens in the internal token stores and the links, by pushing a token into the external store, and can increase it by pulling a token from the external store.

We use the composite atomicity model with link registers. An algorithm $\mathcal{A}$ is specified by a set of local variables in processes, a set of shared variables in link registers, and actions that specify how a process v updates its local variables and shared variables in its output registers at each step, but not the local variables of its neighbors. When a process v executes the procedure, according to the values of its local variables, shared variables in $r_{u,v}$ and $r_{v,u}$ for all $u \in N(v)$, the number of tokens in its token store (i.e., $|v.\texttt{tokenStore}|$), and two given parameters ℓ and k, process v updates its local variables and shared variables of $r_{v,u}$, and executes `Push` and `Pull` arbitrarily many times. If a process v executes $v.\texttt{Push}(x)$, it append a token with bit sequence x into its token store; when it executes $v.\texttt{Pull}()$, it extracts and removes an arbitrary token from its token store. A process v can detect whether its local token store is empty (i.e., $|v.\texttt{tokenStore}| = 0$), and whether its local token store is full (i.e., $|v.\texttt{tokenStore}| = \ell$). A process v is not allowed to invoke `Pull` when its token store is empty, nor to invoke `Push` when its token store is full. The root process v_{root} can push a token into or pull a token out of the external store by executing $\texttt{exPush}(x)$ and $\texttt{exPull}()$, respectively. Typically, it can move a token from its local token store to the external store by $\texttt{exPush}(\texttt{Pull}())$ and can move a token in the reverse direction by $\texttt{Push}(\texttt{exPull}())$. There is one restriction, however, each of $\texttt{exPush}()$ and $\texttt{exPull}()$ can be executed at most once per step.[1] We say a process is *enabled* if execution of the procedure would change the value of at least one variable; otherwise, the process is *disabled*.

The values of all variables in a process define the *state* of the process; similarly, the values of the variables, including `token`, in a link register define the state of that register. A global state or *configuration* of T is specified by the the states of all processes, the states of all link registers, and the contents of the token stores of all processes. Let γ and γ' be two configurations of an algorithm $\mathcal{A}$ on T. We say that $\gamma \mapsto \gamma'$ is a *step* of $\mathcal{A}$ if there is a non-empty set $S \subseteq V$ of enabled processes such that γ changes to γ' when all the processes in S simultaneously execute the procedure of $\mathcal{A}$. A process in S is said to be *selected* at the step $\gamma \mapsto \gamma'$. We define an *execution* of $\mathcal{A}$ to be a maximal sequence $\gamma_0, \gamma_1, \dots$ of configurations such that each $\gamma_i \mapsto \gamma_{i+1}$ is a step of $\mathcal{A}$. For any local variable `var1` and any shared variable `var2`, we denote local variable `var1` of a process v by $v.\texttt{var1}$, and shared variable `var2` of register $r_{u,v}$ by $r_{u,v}.\texttt{var2}$. Furthermore, for any configuration γ, we denote the values of $v.\texttt{var1}$ and $r_{u,v}.\texttt{var2}$ in configuration γ by $\gamma(v).\texttt{var1}$ and $\gamma(r_{u,v}).\texttt{var2}$, respectively.

We assume that a *scheduler* (or *daemon*) selects a set of processes that execute at each step of an execution. The execution ends when there are no enabled processes. In this paper, we assume the *distributed unfair daemon*, which selects an arbitrary nonempty subset of enabled processes at each step; it is *unfair* because it is not required to select a specific enabled process v unless v is the only enabled process.

2.2 Problem Specification

We now formally specify the problem. Given positive integers $k \leq \ell$, our goal is to reach a configuration where every process holds exactly k tokens starting from any configuration where the number of tokens at each process is arbitrary in the range $0, 1, \dots, \ell$.

First, we give the definition of *legality* of an algorithm. Intuitively, we want to guarantee that no tokens in the network disappear from the network unless v_{root} pushes them into the external store, and that no new token appears in the network until v_{root} pulls a token from

[1] This is a natural restriction since we have the rule that a process can send at most one token to a given neighbor in a step, as we shall see later. Simply think of the external store as located in a fictitious neighboring process.

the external store. The nature of a token is defined by the application; we can assume it is a bit string. To transfer a token x from process u to v, process u writes x to $r_{u,v}$.token. But v cannot delete x from $r_{u,v}$.token since $r_{u,v}$ is an input register of v. Instead, we assume that every algorithm $\mathcal{A}$ specifies a predicate $\mathcal{P}_{\mathcal{A}}$ which indicates whether or not $\mathcal{A}$ recognizes that a token exists in a link register $r_{u,v}$ according to the states of $r_{u,v}$ and $r_{v,u}$. Let $R_{\mathcal{A}}$ be the set of all the states of a register in algorithm $\mathcal{A}$. Given $\mathcal{P}_{\mathcal{A}} : R_{\mathcal{A}} \times R_{\mathcal{A}} \to \{\mathit{false}, \mathit{true}\}$, we consider that a token represented by the bit sequence x exists in register $r_{u,v}$ if and only if $r_{u,v}$.token $= x$ and $\mathcal{P}_{\mathcal{A}}(a, b) = \mathit{true}$ where a and b are the states of $r_{u,v}$ and $r_{v,u}$, respectively. Then, the multiset of tokens in the network with a configuration γ is uniquely determined and we denote this multiset by $m_{\gamma,\mathcal{P}_{\mathcal{A}}}$. (i.e., A bit sequence x appears n_x times in multiset $m_{\gamma,\mathcal{P}_{\mathcal{A}}}$ if and only if exactly n_x tokens with contents x exists in the network.) We say that a step $\gamma \mapsto \gamma'$ is *legal* with $\mathcal{P}_{\mathcal{A}}$ if the following three conditions hold: (i) process v_{root} does not invoke both the two commands exPush(x) and exPull() in step $\gamma \mapsto \gamma'$, (ii) if v_{root} pulls a token x from (resp. pushes a token x to) the external store in step $\gamma \mapsto \gamma'$, the number of tokens with contents x increases (resp. decreases) by one and the numbers of other tokens are unchanged from $m_{\gamma,\mathcal{P}_{\mathcal{A}}}$ to $m_{\gamma',\mathcal{P}_{\mathcal{A}}}$, and (iii) if v_{root} does not execute either exPush(x) nor exPull() during step $\gamma \mapsto \gamma'$, then $m_{\gamma,\mathcal{P}_{\mathcal{A}}} = m_{\gamma',\mathcal{P}_{\mathcal{A}}}$. We say that an algorithm $\mathcal{A}$ is legal with $\mathcal{P}_{\mathcal{A}}$ if every step of algorithm $\mathcal{A}$ is legal with $\mathcal{P}_{\mathcal{A}}$.

We say that $\mathcal{A}$ is a self-stabilizing [6] token distribution algorithm if $\mathcal{A}$ is legal with some predicate $\mathcal{P}_{\mathcal{A}}$ and there is a set $\mathcal{L}_{\mathcal{A}}$ of *legitimate* configurations, such that the following three conditions are satisfied: (i) **Closure:** If $\gamma \in \mathcal{L}_{\mathcal{A}}$ holds, and $\gamma \mapsto \gamma'$ is a step of $\mathcal{A}$, then $\gamma' \in \mathcal{L}_{\mathcal{A}}$. (ii) **Convergence:** Every execution of $\mathcal{A}$, which starts from an arbitrary configuration, contains a configuration in $\mathcal{L}_{\mathcal{A}}$. (iii) **Correctness:** At any $\gamma \in \mathcal{L}_{\mathcal{A}}$, every process holds exactly k tokens in its token store and no registers hold tokens in γ in the sense of predicate $\mathcal{P}_{\mathcal{A}}$. We also say that a self-stabilizing token distribution algorithm $\mathcal{A}$ is *silent* if every execution of $\mathcal{A}$ is finite, that is, reaches a configuration where no process is enabled.

2.3 Complexities

We evaluate token distribution algorithms with three metrics – time complexity, space complexity, and the number of token moves.

We measure the time complexity in terms of (asynchronous) *rounds*. Let $\Gamma = \gamma_0, \gamma_1, \ldots$ be an execution of $\mathcal{A}$, and let $\mathcal{E}$ be the set of processes which are enabled at γ_0. We define the *first round* of execution Γ to be the smallest prefix, say $\gamma_0, \ldots, \gamma_t$, of Γ such that every member of $\mathcal{E}$ either executes or becomes disabled during the first t steps. We define $\gamma_0, \ldots \gamma_t$ to be the *range* of the first round of Γ. Let Γ' be the suffix of Γ starting from γ_t. The second round of Γ is defined to be the first round of Γ', and so forth.

It requires lb bits (resp. b bits) of space to manage a token store in each process (resp. variable token in each register). In this paper, we focus on *work space complexity* in each process and in each register of an algorithm. The work space complexity in each process (resp. in each register) is the bit length to represent all variables on the process (resp. in the register) except for tokenStore (resp. token).

Generally, a token is transferred from a process u to a process v in the following two steps: (i) u moves the token from u.tokenStore to $r_{u,v}$.token; (ii) v moves the token from $r_{u,v}$.token to v.tokenStore. In this paper, we regard the above two steps together as one token move and consider the number of token moves as the number of the occurrences of the former steps. Specifically, we say that a token moves from u to v in step $\gamma \mapsto \gamma'$ of algorithm $\mathcal{A}$ when predicate $\neg\mathcal{P}_{\mathcal{A}}(\gamma(r_{u,v}), \gamma(r_{v,u})) \wedge \mathcal{P}_{\mathcal{A}}(\gamma'(r_{u,v}), \gamma'(r_{v,u}))$ holds. Furthermore, introducing a *virtual process* $p(v_{\mathrm{root}}) \notin V$, we also say that a token moves from v_{root} to

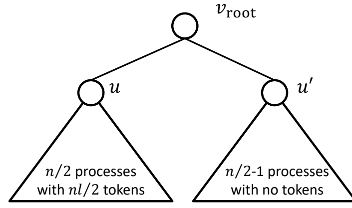


Figure 1 A tree to prove Theorem 1.

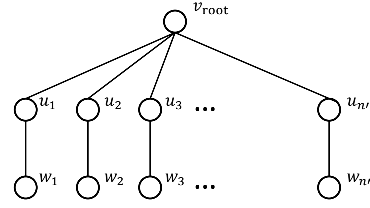


Figure 2 A tree to prove Theorem 2.

$p(v_{\text{root}})$ (resp. from $p(v_{\text{root}})$ to v_{root}) in a step when v_{root} invokes `exPush()` (resp. `exPull()`) in that step. We are interested in the number of redundant token moves. Given $v \in V$ and configuration γ, let $\tau(\gamma, v)$ be the number of tokens in input registers of v in configuration γ. Then, we define $\Delta(\gamma, v) = \sum_{u \in T_v} d(\gamma, u)$ where $d(\gamma, u) = |\gamma(u).\texttt{tokenStore}| + \tau(\gamma, u) - k$. Intuitively, $\Delta(\gamma, v)$ is the number of tokens that v must send to $p(v)$ to achieve the token distribution in an execution starting from configuration γ if $\Delta(\gamma, v) \geq 0$; Otherwise, $p(v)$ must send $-\Delta(\gamma, v)$ tokens to v in the execution. Therefore, in an execution $\Gamma = \gamma_0, \gamma_1, \ldots,$ we need at least $\sum_{v \in V} |\Delta(\gamma_0, v)|$ token moves to achieve token distribution. We define *the number of redundant token moves* in $\Gamma = \gamma_0, \gamma_1, \ldots$ as the total number of token moves in the execution minus $\sum_{v \in V} |\Delta(\gamma_0, v)|$.

3 Lower Bounds

▶ **Theorem 1.** *For any self-stabilizing token distribution algorithm $\mathcal{A}$, there exists an execution of $\mathcal{A}$ that takes $\Omega(nl)$ rounds to reach a configuration where all processes hold exactly k tokens.*

Proof. Consider a tree network of an even number of processes where v_{root} has two children u and u' where $n_u = n/2$ and $n_{u'} = n/2 - 1$, and consider a configuration of $\mathcal{A}$ on the graph where T_u holds $n_u \cdot l$ tokens in total and $T_{u'}$ holds zero tokens (Figure 1). Obviously, process u must be selected by the scheduler $\Omega(n(l-k))$ times to send $n_u(l-k)$ tokens to v_{root} and process u' must be selected by the scheduler $\Omega(nk)$ times to receive $n_v \cdot k$ tokens from v_{root}. Thus, there exists an execution of $\mathcal{A}$ starting from this configuration which takes $\max(\Omega(n(l-k)), \Omega(nk)) = \Omega(nl)$ rounds to achieve the token distribution. ◀

▶ **Theorem 2.** *For any self-stabilizing token distribution algorithm $\mathcal{A}$, there exists an execution of $\mathcal{A}$ such that the number of redundant token moves in the execution is $\Omega(n)$.*

Proof. Consider the tree network where v_{root} has n' children $u_1, u_2, \ldots, u_{n'}$, each u_i has exactly one child w_i, and no other processes exist (See Figure 2). Consider an execution of $\mathcal{A}$, on this graph, that starts from a configuration where, for all $i = 1, 2, \ldots, n'$, process u_i holds $k+1$ tokens, w_i holds no token, and u_i, $r_{u_i, v_{\text{root}}}$, r_{v_{root}, u_i}, r_{u_i, w_i}, and r_{w_i, u_i} are in states such that u_i sends a token to v_{root} when u_i is selected by the scheduler. (Such states must exist for any self-stabilizing token distribution algorithm.) If every u_i is selected by the scheduler in the first step of this execution, then $n' = \lfloor n/2 \rfloor$ redundant token moves must happen in the first step. ◀

4 Constant-Space Algorithms for Self-stabilizing Token Distribution

Before presenting the three algorithms – *Base*, *SyncTokenDist*, and *PIFTokenDist* – we describe the token-passing mechanism (or handshake mechanism) that all these algorithms use to send and receive a token between two processes. Each register $r_{u,v}$ has two shared boolean variables $r_{u,v}.\texttt{exist}$ and $r_{u,v}.\texttt{ready}$ in addition to $r_{u,v}.\texttt{token}$. We use predicate $\mathcal{P}_{Base}(a, b) \equiv a.\texttt{exist} \wedge b.\texttt{ready}$ (where a and b are states of $r_{u,v}$ and $r_{v,u}$, respectively) for all the three algorithms to represent the existence of a token in link register: register $r_{u,v}$ holds a token when $r_{u,v}.\texttt{exist} \wedge r_{v,u}.\texttt{ready}$ holds. Let u and v be neighboring processes. We define four predicates used in the pseudocodes as follows:

$$\begin{aligned} \mathit{IsReady}(u,v) &\equiv (\neg r_{u,v}.\texttt{exist} \wedge r_{v,u}.\texttt{ready}),\\ \mathit{JustSent}(u,v) &\equiv (r_{u,v}.\texttt{exist} \wedge r_{v,u}.\texttt{ready}),\\ \mathit{JustReceived}(u,v) &\equiv (r_{u,v}.\texttt{exist} \wedge \neg r_{v,u}.\texttt{ready}),\\ \mathit{NotReady}(u,v) &\equiv (\neg r_{u,v}.\texttt{exist} \wedge \neg r_{v,u}.\texttt{ready}). \end{aligned}$$

All the three algorithms adopt the token passing mechanism consisting of the following rules to send a token from a process u to a process v:

Rule 1 When $\mathit{IsReady}(u, v)$ and $|u.\texttt{tokenStore}| > 0$, process u may perform "$r_{u,v}.\texttt{token} \leftarrow \texttt{Pull}()$" and "$r_{u,v}.\texttt{exist} \leftarrow \mathit{true}$". After that, $\mathit{JustSent}(u, v)$ becomes true.

Rule 2 When $\mathit{JustSent}(u, v)$ and $|v.\texttt{tokenStore}| < l$, process v may perform "$\texttt{Push}(r_{u,v}.\texttt{token})$" and "$r_{v,u}.\texttt{ready} \leftarrow \mathit{false}$". After that, $\mathit{JustReceived}(u, v)$ becomes true.

Rule 3 When $\mathit{JustReceived}(u, v)$, process u may perform "$r_{u,v}.\texttt{exist} \leftarrow \mathit{false}$". After that, $\mathit{NotReady}(u, v)$ becomes true.

Rule 4 When $\mathit{NotReady}(u, v)$, process v may perform "$r_{v,u}.\texttt{ready} \leftarrow \mathit{true}$". After that, $\mathit{IsReady}(u, v)$ becomes true.

By definition of $\mathcal{P}_{Base}$, a token exists in $r_{u,v}$ if and only if $\mathit{JustSent}(u, v)$ is true. In any step $\gamma \mapsto \gamma'$, none of the above rule changes $m_{\gamma,\mathcal{P}_{Base}}$, i.e., a multiset of tokens in the network. In addition to the above rules, there is another mechanism to update a local token store. The root process v_{root} pushes a token to and pulls a token from the external store. Specifically, v_{root} pulls a token from the external store by $\texttt{Push}(\texttt{exPull}())$, and v_{root} pushes a token to the external store by "$\texttt{exPush}(\texttt{Pull}())$". The former (resp. latter) operation is performed only when $|v_{\mathrm{root}}.\texttt{tokenStore}| < l$ (resp. $|v_{\mathrm{root}}.\texttt{tokenStore}| > 0$). Shared variables `token`, `exist`, and `ready` are never updated in any way other than Rules 1-4.

An arbitrary step $\gamma \mapsto \gamma'$ of an algorithm which adopts the above token mechanism is legal because Rules 1-4 do not affect the multiset $m_{\gamma',\mathcal{P}_{Base}}$. Thus the following lemma holds.

▶ **Lemma 3.** *Algorithms Base, SyncTokenDist, and PIFTokenDist are all legal with $\mathcal{P}_{Base}$.*

4.1 Algorithm *Base*

In this section, we define our self-stabilizing token distribution algorithm, which we call *Base*. The work space complexity of *Base* in process (resp. on register) is zero (resp. constant). Its convergence time is $O(n\ell)$ rounds and it makes $\Theta(nh\epsilon)$ redundant token moves.

Some functions take a configuration (e.g., γ) as one of their arguments, such as $\Delta(\gamma, v)$, defined in Section 2.2. In what follows, we omit the configuration when it is clear from the context. For example, we write $\Delta(v)$ instead of $\Delta(\gamma, v)$. We use the standard notation $\mathrm{sgn}(x)$, that is, $\mathrm{sgn}(x) = 1$, $\mathrm{sgn}(x) = 0$, and $\mathrm{sgn}(x) = -1$ if $x > 0$, $x = 0$, and $x < 0$, respectively.

Algorithm 1 *Base*

[Variables of process v and register $r_{v,u}$]

v.tokenStore {an array containing at most l tokens}

$r_{v,u}$.exist, $r_{v,u}$.ready

$r_{v,u}$.est $\in \{1, 0^+, 0, 0^-, -1, \perp\}$ {$r_{v,u}$.est $\in \{1, 0, -1\}$ if v is a leaf.}

$r_{v,u}$.token $\in 2^b$

[Actions of process v]

```
 1: r_{v,u}.exist ← false for all u ∈ N(v) such that JustReceived(v, u)
 2: r_{v,u}.ready ← true for all u ∈ N(v) such that NotReady(u, v)
 3: ReceiveToken(u) for all u ∈ N(v) such that JustSent(u, v)
 4: SendToken(u) for all u ∈ C(v) such that r_{u,v}.est = −1 ∧ IsReady(v, u)
 5: if v = v_root then
 6:     AdjustTokens()
 7: else
 8:     SendToken(p(v)) if Est(v) = 1 ∧ IsReady(v, p(v))
 9:     r_{v,p(v)}.est ← Est(v)
10: end if
```

[SendToken(u)]:

```
11: if |v.tokenStore| > 0 then
12:     r_{v,u}.token ← Pull()
13:     r_{v,u}.exist ← true
14:     ReceiveToken(w) if ∃w ∈ N(v) : JustSent(w, v)
15: end if
```

[ReceiveToken(u)]:

```
16: if |v.tokenStore| < l then
17:     Push(r_{u,v}.token)
18:     r_{v,u}.ready ← false
19: end if
```

[AdjustTokens()]:

```
20: if Est(v) = −1 then
21:     Push(exPull())
22: else if Est(v) = 1 then
23:     exPush(Pull())
24:     ReceiveToken(u) if ∃u ∈ N(v) : JustSent(u, v)
25: end if
```

During an execution of *Base*, each process v tries to estimate $\mathrm{sgn}(\Delta(v))$, that is, tries to find whether $\Delta(v)$ is positive, negative, or just zero. Each process $v \neq v_{\mathrm{root}}$ reports that estimate to its parent $p(v)$ using a shared variable $r_{v,p(v)}$.est. When its estimate is negative, $p(v)$ sends a token to v if $p(v)$ holds a token and $\mathit{IsReady}(p(v), v)$. When the estimate is positive, v sends a token to its parent $p(v)$ if v holds a token and $\mathit{IsReady}(v, p(v))$. The root v_{root} always pulls a new token from the external store to increase $\Delta(v_{\mathrm{root}})$ when its estimate is negative, and pushes a token to the external store to decrease $\Delta(v_{\mathrm{root}})$ when the estimate is positive. If all processes v correctly estimate $\mathrm{sgn}(\Delta(v))$, each of them eventually holds k tokens. After that, no process sends a token.

Thus, estimating $\mathrm{sgn}(\Delta(v))$ is the key of algorithm *Base*. Each process v computes $Est(v)$, its estimate of $\mathrm{sgn}(\Delta(v))$ as follows.

$$Est(v) = \begin{cases} 1 & (Diff(v) > 0 \land \forall u \in C(v) : r_{u,v}.\texttt{est} \in \{1, 0^+, 0\}) \\ 0^+ & (Diff(v) = 0 \land \forall u \in C(v) : r_{u,v}.\texttt{est} \in \{1, 0^+, 0\} \\ & \qquad \land \exists w \in C(v) : r_{u,v}.\texttt{est} \in \{0^+, 1\}) \\ 0 & (Diff(v) = 0 \land \forall u \in C(v) : r_{u,v}.\texttt{est} = 0) \\ 0^- & (Diff(v) = 0 \land \forall u \in C(v) : r_{u,v}.\texttt{est} \in \{-1, 0^-, 0\} \\ & \qquad \land \exists w \in C(v) : r_{u,v}.\texttt{est} \in \{0^-, -1\}) \\ -1 & (Diff(v) < 0 \land \forall u \in C(v) : r_{u,v}.\texttt{est} \in \{-1, 0^-, 0\}) \\ \bot & (\text{otherwise}), \end{cases}$$

$$Diff(v) = |v.\texttt{tokenStore}| + |\{u \in N(v) \mid JustSent(u, v)\}| - k$$

where the candidate values 1, 0^+, 0, 0^-, -1, and $\bot$ of $Est(v)$ represent that the estimate is positive, "never negative", zero, "never positive", negative, and "unsure", respectively. A process sends a token to its parent only when its estimate is 1, and it sends a token to its child only when the child's estimate is -1. Note that $\Delta(v) = \sum_{u \in T_v} Diff(u)$. For process $v \in V \setminus \{v_{\mathrm{root}}\}$, the domain of variable $r_{v,p(v)}.\texttt{est}$ is $\{1, 0^+, 0, 0^-, -1\}$ if v is not a leaf, and is $\{1, 0, -1\}$ if v is a leaf.

Algorithm 1 gives the pseudocode of *Base*. When v executes, it first updates $r_{v,u}.\texttt{exist}$ and $r_{v,u}.\texttt{ready}$ for all $u \in N(v)$ such that $JustReceived(v, u)$ or $NotReady(u, v)$, according to the token-passing mechanism (Lines 1-2). Then, v checks whether each input register $r_{u,v}$ holds a token, and receives that token by invoking $\texttt{ReceiveToken}(u)$ if its token store is not full, using the token-passing mechanism (Lines 3, 16-19). Then, v sends a token to each $u \in C(v)$ such that $IsReady(v, u)$ and $r_{u,v}.\texttt{est} = -1$ (i.e., $\Delta(u)$ is estimated to be negative) by invoking $\texttt{SendToken}(u)$ if v's token store is not empty (Lines 4, 11-15). Next, $v \in V \setminus \{v_{\mathrm{root}}\}$ and v_{root} perform different action. If $v \neq v_{\mathrm{root}}$, it sends a token to its parent if $IsReady(v, p(v))$ and $Est(v) = 1$, i.e., $\Delta(v)$ is estimated to be positive, and reports the latest estimate of $\mathrm{sgn}(\Delta(v))$ to its parent, and stores $Est(v)$ into $r_{v,p(v)}.\texttt{est}$ (Lines 7-10). Note that $Diff(v)$ decreases when v sends a token to its parent, hence the value of $Est(v)$ may differ in Lines 8 and 9. Process v_{root} invokes $\texttt{AdjustTokens}()$ by which v_{root} may increase or decrease the number of tokens in the tree by pulling a token from or pushing a token to the external store (Lines 6, 20-25).

In the algorithm description, we have used several functions which take a process as an argument, like $Est(v)$ and $Diff(v)$. The values of such functions depend on the states of the process and its registers, i.e., they depend on the current configuration. In what follows, we sometimes denote such functions with an explicitly specified configuration. For example, we sometimes denote $Est(v)$ in configuration γ by $Est(\gamma, v)$ and denote $Diff(v)$ in configuration γ by $Diff(\gamma, v)$.

In what follows, we show the correctness, the number of redundant token moves, and the convergence time of algorithm *Base*. First, we show that every execution of *Base* is finite (Lemma 4). The correctness of *Base* immediately follows. (Theorem 5).

▶ **Lemma 4.** *Every execution of Base is finite.*

Proof. Fix an execution $\Gamma = \gamma_0, \gamma_1, \ldots$ of *Base*. For any $v \in V$, we define a predicate $P_{\mathrm{finite}}(v) \equiv P_{\mathrm{finiteMove}}(v) \land P_{\mathrm{finiteEst}}(v)$, where $P_{\mathrm{finiteMove}}(v)$ and $P_{\mathrm{finiteEst}}(v)$ are also predicates: $P_{\mathrm{finiteMove}}(v)$ holds if and only if v sends and receives tokens only finitely many times in Γ, and $P_{\mathrm{finiteEst}}(v)$ holds if and only if $v = v_{\mathrm{root}}$ or v changes the value

of $r_{v,p(v)}.\texttt{est}$ only finitely many times in Γ. In the remainder of this proof, we prove $(\forall u \in C(v); P_{\text{finite}}(u)) \Rightarrow P_{\text{finite}}(v)$ for all $v \in V$. This proposition guarantees that (i) every leaf w satisfies $P_{\text{finite}}(w)$ because it has no children, and (ii) every process v with height $i > 0$ satisfies $P_{\text{finite}}(v)$ if every process u with height $i-1$ satisfies $P_{\text{finite}}(u)$. Thus, $P_{\text{finite}}(v')$ holds for all v by induction on the height h_v of v, from which gives the lemma.

Let $v \in V$, and suppose $P_{\text{finite}}(u)$ holds for all $u \in C(v)$. Then, v sends or receives no token to or from its children, and $r_{u,v}.\texttt{est}$ remains unchanged for all $u \in C(v)$ after some point of the execution. Let $\gamma_t \mapsto \gamma_{t+1}$ be the first step that v is selected after that point. (The unfair daemon may never selects v after that point, but we need not consider this case because then $P_{\text{finite}}(v)$ clearly holds.) In an execution after that step, say $\Gamma_{t+1} = \gamma_{t+1}, \gamma_{t+2}, \dots$, process v sends a token to $p(v)$ only if $\mathit{Diff}(v) > 0$, and $p(v)$ sends a token to v only if $\mathit{Diff}(v) < 0$. Hence, v sends and receives tokens at most $|\mathit{Diff}(\gamma_{t+1}, v)|$ times in Γ_{t+1}, which implies $P_{\text{finiteMove}}(v)$. Thus, there exists $t' > t$ such that v never sends or receives a token after $\gamma_{t'}$. Since $\mathit{Diff}(v)$ never changes after $\gamma_{t'}$, $\mathit{Est}(v)$ also never changes after $\gamma_{t'}$. This means that $r_{v,p(v)}.\texttt{est}$ never changes after $\gamma_{t'+1}$, which implies $P_{\text{finiteEst}}(v)$. Thus $P_{\text{finite}}(v)$ holds in all cases. ◀

▶ **Theorem 5.** *Algorithm Base is a silent self-stabilizing token distribution algorithm.*

Proof. Let $\Gamma = \gamma_0, \gamma_1, \dots$ be an execution of *Base*. Lemma 4 guarantees that γ ends at its final configuration γ_f, that is, $\Gamma = \gamma_0, \gamma_1, \dots, \gamma_f$. No process is enabled in γ_f. Hence, every process holds k tokens in its token store and no token exists in registers in γ_f because otherwise some process must be enabled. ◀

We now give an asymptotically tight bound. on the number of redundant token moves of *Base*. Recall that $\epsilon = \min(k, l-k)$.

▶ **Lemma 6.** *The number of redundant token moves in any execution of Base is* $O(nh\epsilon)$.

Proof. For any process v, let $\Delta^+(v) = \sum_{u \in T_v} \max(\mathit{Diff}(u), 0)$ and $\Delta^-(v) = \sum_{u \in T_v} \max(-\mathit{Diff}(u), 0)$. During the execution $\Gamma = \gamma_0, \gamma_1, \dots$ of *Base*, each v sends a token to $p(v)$ at most $\Delta^+(\gamma_0, v) + n_v$ times. This is because $\Delta^+(v)$ is monotonically non-increasing, except for the case that the parent of a process $u \in T_v$ sends a token to u before the first time u is selected by the scheduler, and $\Delta^+(v)$ decrements by one every time v sends a token to $p(v)$. Similarly, $p(v)$ sends a token to v at most $\Delta^-(\gamma_0, v)$ times in Γ. Since $\Delta(\gamma_0, v) = \Delta^+(\gamma_0, v) - \Delta^-(\gamma_0, v)$ and $\sum_{v \in V} n_v \le nh$, the number of redundant token moves in Γ is at most $\sum_{v \in V}(\Delta^+(\gamma_0, v) + \Delta^-(\gamma_0, v) + n_v) - \sum_{v \in V} |\Delta(\gamma_0, v)| = \sum_{v \in V}(2\min(\Delta^+(\gamma_0, v), \Delta^-(\gamma_0, v)) + n_v) \le \sum_{v \in V}(2\min(n_v(l-k), n_v \cdot k) + n_v) = O(nh\epsilon)$. ◀

▶ **Lemma 7.** *The number of redundant token moves in an execution of Base is* $\Omega(nh\epsilon)$.

Proof. Consider the network shown in Figure 3 where $2x \cdot 2y$ processes except for v_{root} are divided into four sets A, B, C, and D, each of which consists of xy processes. Consider a configuration where each process v in $\mathrm{A} \cup \mathrm{D}$ holds no token (i.e., $\mathit{Diff}(v) = -k$), each process u in $\mathrm{B} \cup \mathrm{C}$ holds ℓ tokens (i.e., $\mathit{Diff}(u) = \ell - k$), and $r_{w_i,p(w_i)}.\texttt{est} = r_{w'_i,p(w'_i)} = 0$ holds for all $i = 1, 2, \dots, y$. In an execution starting from the configuration, the scheduler can select processes only in $\mathrm{A} \cup \mathrm{D} \cup \{v_{\text{root}}\}$ until all processes in $\mathrm{A} \cup \mathrm{D} \cup \{v_{\text{root}}\}$ become disabled. Such an execution must make at least $\min(x(x-1)yk/2, x(x-1)y(l-k)/2) = \Omega(nh\epsilon)$ redundant token moves. ◀

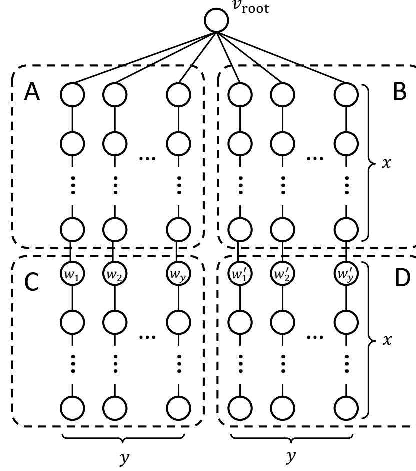


Figure 3 A tree to prove Lemma 7.

Finally, we analyze the convergence time of *Base*. We first consider the values of $r_{v,p(v)}.\texttt{est}$. Specifically, we prove that every execution of *Base* satisfies predicate $P_{\text{est}}(v)$, which is defined as follows, within $2h_v$ rounds, for any process v other than v_{root}:

$$\begin{aligned}
P_{\text{est}}(v) &\equiv P_1(v) \wedge P_2(v) \wedge P_3(v) \wedge P_4(v) \wedge Q_1(v) \wedge Q_2(v) \wedge Q_3(v) \wedge \forall u \in C(v) : P_{\text{est}}(u)\\
P_1(v) &\equiv r_{v,p(v)}.\texttt{est} \in \{1, 0^+\} \;\Rightarrow\; \Delta(v) > 0 \wedge \mathit{Diff}(v) \geq 0\\
P_2(v) &\equiv r_{v,p(v)}.\texttt{est} = 0 \;\Rightarrow\; \Delta(v) = 0 \wedge \mathit{Diff}(v) = 0\\
P_3(v) &\equiv r_{v,p(v)}.\texttt{est} = 0^- \;\Rightarrow\; \Delta(v) \leq 0 \wedge \mathit{Diff}(v) \leq 0\\
P_4(v) &\equiv r_{v,p(v)}.\texttt{est} = -1 \;\Rightarrow\; (\Delta(v) < 0 \vee (\Delta(v) = 0 \wedge \mathit{JustSent}(p(v), v)))\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge\, \mathit{Diff}(v) \leq 0,\\
Q_1(v) &\equiv r_{v,p(v)}.\texttt{est} \in \{1, 0^+\} \;\Rightarrow\; \forall u \in T_v : r_{u,p(u)}.\texttt{est} \in \{1, 0^+, 0\},\\
Q_2(v) &\equiv r_{v,p(v)}.\texttt{est} = 0 \;\Rightarrow\; \forall u \in T_v : r_{u,p(u)}.\texttt{est} = 0,\\
Q_3(v) &\equiv r_{v,p(v)}.\texttt{est} \in \{0^-, -1\} \;\Rightarrow\; \forall u \in T_v : r_{u,p(u)}.\texttt{est} \in \{0, 0^-, -1\}.
\end{aligned}$$

▶ **Lemma 8.** *Let $v \in V \setminus \{v_{\text{root}}\}$ and let $\gamma \mapsto \gamma'$ be a step of Base where $P_{\text{est}}(\gamma, v)$ holds. Then, the following three statements hold:*

$$\gamma(r_{v,p(v)}).\texttt{est} = 0 \Rightarrow \gamma'(r_{v,p(v)}).\texttt{est} = 0 \tag{1}$$

$$\gamma(r_{v,p(v)}).\texttt{est} \in \{1, 0^+\} \Rightarrow \gamma'(r_{v,p(v)}).\texttt{est} \in \{1, 0^+, 0\} \tag{2}$$

$$\gamma(r_{v,p(v)}).\texttt{est} \in \{0^-, -1\} \Rightarrow \gamma'(r_{v,p(v)}).\texttt{est} \in \{0, 0^-, -1\} \tag{3}$$

Proof. If $\gamma(r_{v,p(v)}).\texttt{est} = 0$, then $\gamma'(r_{v,p(v)}).\texttt{est} = 0$ because $P_{\text{est}}(\gamma, v)$ implies that $\mathit{Diff}(\gamma, v) = 0$ and $\gamma(r_{u,v}).\texttt{est} = 0$ for all $u \in C(v)$. If $\gamma(r_{v,p(v)}).\texttt{est} \in \{1, 0^+\}$, then $\gamma'(r_{v,p(v)}).\texttt{est} \in \{1, 0^+, 0\}$ because $P_{\text{est}}(\gamma, v)$ implies that $\mathit{Diff}(\gamma, v) \geq 0$ and $\forall u \in C(v) : \gamma(r_{u,v}).\texttt{est} \in \{1, 0^+, 0\}$. Similarly, $\gamma'(r_{v,p(v)}).\texttt{est} \in \{0, 0^-, -1\}$ holds if $\gamma(r_{v,p(v)}).\texttt{est} \in \{0^-, -1\}$. ◀

▶ **Lemma 9.** *Let $v \in V$. If $P_{\text{est}}(u)$ holds for all $u \in C(v)$, then once $P_{\text{est}}(v)$ holds, $P_{\text{est}}(v)$ always holds.*

Proof. Let $\gamma \mapsto \gamma'$ be a step of *Base* where $P_{\text{est}}(\gamma, v)$ holds and $P_{\text{est}}(\gamma, u)$ and $P_{\text{est}}(\gamma', u)$ hold for all $u \in C(v)$. It suffices to show $P_{\text{est}}(\gamma', v)$ to prove the lemma. First, consider the case of $\gamma(r_{v,p(v)}).\texttt{est} = \bot$. If $r_{v,p(v)}.\texttt{est}$ remains $\bot$ in step $\gamma \mapsto \gamma'$, then $P_{\text{est}}(\gamma', v)$ trivially holds. If $r_{v,p(v)}.\texttt{est}$ becomes 1 or 0^+ at the step, i.e., $\gamma'(r_{v,p(v)}.\texttt{est}) \in \{1, 0^+\}$, then $P_1(\gamma', v)$ holds and $\gamma(r_{u,v}.\texttt{est}) \in \{1, 0^+, 0\}$ for all $u \in C(v)$ by the definition of *Base*. The latter statement, and Lemma 8, imply $Q_1(\gamma', v)$. Hence, $P_{\text{est}}(\gamma', v)$ holds if $\gamma'(r_{v,p(v)}.\texttt{est}) \in \{1, 0^+\}$. Similarly, $P_{\text{est}}(\gamma', v)$ holds if $\gamma'(r_{v,p(v)}.\texttt{est}) \in \{0, 0^-, -1\}$. Next, suppose $\gamma(r_{v,p(v)}).\texttt{est} \neq \bot$. In this case, $\gamma'(r_{v,p(v)}).\texttt{est} \neq \bot$ holds by Lemma 8. If $\gamma'(r_{v,p(v)}).\texttt{est} \in \{1, 0^+\}$, then $P_1(\gamma', v)$ trivially holds by the definition of *Base*, and $P_{\text{est}}(\gamma, v)$ and Lemma 8 imply that $\gamma'(r_{u,v}).\texttt{est} \in \{1, 0^+, 0\}$ for all $u \in C(v)$, which implies $Q_1(\gamma', v)$. Hence, $P_{\text{est}}(\gamma', v)$ holds if $\gamma'(r_{v,p(v)}.\texttt{est}) \in \{1, 0^+\}$. Similarly, $P_{\text{est}}(\gamma', v)$ holds if $\gamma'(r_{v,p(v)}.\texttt{est}) \in \{0, 0^-, -1\}$. ◀

▶ **Lemma 10.** *Let $v \in V$. If $P_{\text{est}}(u)$ holds for all $u \in C(v)$, $P_{\text{est}}(v)$ holds once v is selected twice by the scheduler or v is disabled.*

Proof. Consider an execution $\Gamma = \gamma_0, \gamma_1, \dots$ of *Base* where $P_{\text{est}}(\gamma_t, u)$ holds for all $t \geq 0$ and all $u \in C(v)$. During the execution, $P_{\text{est}}(v)$ holds when v is disabled, by the definition of algorithm *Base*, and by the assumption that $P_{\text{est}}(u)$ holds for all $u \in C(v)$. Hence, it suffices to show that $P_{\text{est}}(\gamma_{t+1}, v)$ or $P_{\text{est}}(\gamma_{t'+1}, v)$ holds where $\gamma_t \mapsto \gamma_{t+1}$ and $\gamma_{t'} \mapsto \gamma_{t'+1}$ are the first and the second steps where v is selected by the scheduler in Γ. Due to the token passing mechanism, $p(v)$ cannot send a token to $r_{p(v),v}$ in both steps. For any step $\gamma \mapsto \gamma'$, $P_{\text{est}}(\gamma', v)$ holds by the definition of *Base* if v is selected by the scheduler in $\gamma \mapsto \gamma'$, $p(v)$ cannot send a token to $r_{p(v),v}$ at the step, and $P_{\text{est}}(\gamma, u) \wedge P_{\text{est}}(\gamma', u)$ holds for all $u \in C(v)$. Hence, $P_{\text{est}}(\gamma_{t+1}, v)$ or $P_{\text{est}}(\gamma_{t'+1}, v)$ holds. ◀

▶ **Lemma 11.** *Let $v \in V$ and let $\Gamma = \gamma_0, \gamma_1, \dots$ be an execution of Base. The predicate $P_{\text{est}}(\gamma_t, v)$ holds for any $t \geq s(\Gamma, 2h_v)$, i.e., $P_{\text{est}}(v)$ always holds after $2h_v$ rounds have elapsed.*

Proof. By induction on h_v. ◀

We define $\mathcal{C}_{\text{est}}$ to be the set of all configurations where $P_{\text{est}}(v)$ holds for all $v \neq v_{\text{root}}$. The following corollary directly follows from Lemma 9 and Lemma 10. Furthermore, Lemma 13, below trivially holds by the definition of *Base*. Note that Lemma 13 implies that no redundant token move can occur after an execution reaches a configuration in $\mathcal{C}_{\text{est}}$.

▶ **Corollary 12.** *An execution of Base reaches a configuration of $\mathcal{C}_{\text{est}}$ within $2h$ rounds and never deviates from $\mathcal{C}_{\text{est}}$ thereafter.*

▶ **Lemma 13.** *For any process v, $|\Delta(v)|$ is monotonically non-increasing during an execution of Base starting from a configuration in $\mathcal{C}_{\text{est}}$.*

In the rest of this section, we prove that every execution starting from a configuration in $\mathcal{C}_{\text{est}}$ finishes in $O(n\ell)$ rounds. Given $v \in V$, we define

$$R(v) = \min\left\{c \geq 0 \;\middle|\; \forall j = 1, 2, \dots, |\Delta(v)| : \textstyle\sum_{u \in C_{2(j+c)-1}(v)} |\mathit{Diff}(u)| \geq j)\right\}$$

where $C_i(v)$ is the set of processes in T_v whose distance from v is no greater than i (e.g., $C_0(v) = \{v\}$ and $C_1(v) = \{v\} \cup C(v)$). See Figure 4. In the left tree of the figure, $R(v) = 1$ because $C_{2j+1}(v)$ holds at least j extra tokens (i.e., $\sum_{u \in C_{2j+1}(v)} \mathit{Diff}(v) \geq j$) for all $j = 1, 2, 3$, but C_1 holds no extra token. In the right tree of the figure, $R(v) = 3$ because $C_{2j+5}(v)$ is short at least j tokens (i.e., $\sum_{u \in C_{2j+5}(v)}(-\mathit{Diff}(v)) \geq j$) for $j = 1, 2, 3$, but C_7 is short only one token. Let $\Gamma = \gamma_0, \gamma_1, \dots$ be an execution starting from a

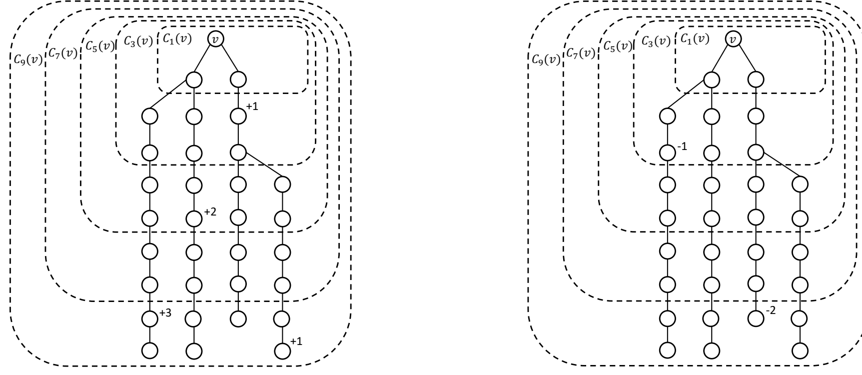


Figure 4 A tree network T_v with $\Delta(v) > 0$ (Left) A tree network with $\Delta(v) < 0$ (Right). The signed integer with process w indicates $\mathit{Diff}(w)$, omitted when $\mathit{Diff}(w) = 0$.

configuration in $\mathcal{C}_{\text{est}}$ and $v \in V \setminus \{v_{\text{root}}\}$ be a process such that $r_{v,p(v)}.\texttt{est} \neq \bot$ holds for some configuration γ_t. Then either $\forall u \in T_v : r_{u,p(u)}.\texttt{est} \in \{1, 0^+, 0\}$ or $\forall u \in T_v : r_{u,p(u)}.\texttt{est} \in \{0, 0^-, -1\}$ holds for $\gamma_t, \gamma_{t+1}, \ldots$. Intuitively, $R(\gamma_t, v)$ has the following meaning: (i) when $\gamma_t(r_{v,p(v)}).\texttt{est} \in \{1, 0^+\}$, v sends $\Delta(\gamma_t, v)$ tokens within $8(R(\gamma_t, v) + \Delta(\gamma_t, v))$ rounds in $\gamma_t, \gamma_{t+1}, \ldots$ if $p(v)$ always receives a token from $r_{v,p(v)}$ immediately after v sends a token to $r_{v,p(v)}$, and (ii) when $\gamma_t(r_{v,p(v)}).\texttt{est} \in \{0^-, -1\}$, v receives $-\Delta(\gamma_t, v)$ tokens within $8(R(\gamma_t, v) - \Delta(\gamma_t, v))$ rounds in $\gamma_t, \gamma_{t+1}, \ldots$ if $p(v)$ always sends a token to $r_{p(v),v}$ immediately after $\mathit{IsReady}(p(v), v) \land r_{p(v),v}.\texttt{est} = -1$ becomes true. For any process $v \in V$, we define $f(v)$ as follows:

$$f(v) = \begin{cases} R(v) + |\Delta(v)| + \left|\{u \in T_v \mid r_{u,p(u)}.\texttt{est} \neq 0\}\right| & (v \neq v_{\text{root}} \land r_{v,p(v)}.\texttt{est} \neq \bot) \\ l + 3 + \sum_{u \in C(v)} f(u) & (v \neq v_{\text{root}} \land r_{v,p(v)}.\texttt{est} = \bot) \\ \sum_{u \in C(v)} f(u) & (v = v_{\text{root}}). \end{cases}$$

▶ **Lemma 14.** *For any configuration γ, $f(\gamma, v_{\text{root}}) = O(n\ell)$.*

Proof. Let v be any process other than v_{root}. Define $V' = \{u \in T_v \mid \gamma(r_{u,p(u)}).\texttt{est} \neq \bot \land (u = v \lor \gamma(r_{p(u),p(p(u))}).\texttt{est} = \bot)\}$. Then, $f(\gamma, v) \leq \sum_{u \in T_v} (\ell + 3) + \sum_{u \in V'} (R(\gamma, u) + |\Delta(\gamma, u)| + |\{u' \in T_u \mid \gamma(r_{u',p(u')}).\texttt{est} \neq 0\}|) \leq n_v(\ell + 3) + \sum_{u \in V'} (h_u + n_u \cdot \ell + n_u) \leq n_v(\ell + 3) + \sum_{u \in V'} n_u(\ell + 2) \leq n_v(2\ell + 5)$. Therefore, $f(\gamma, v_{\text{root}}) \leq \sum_{v \in C(v_{\text{root}})} n_v(2\ell + 5) = O(n\ell)$. ◀

Note that $f(v)$ is monotonically non-increasing during any execution starting from a configuration in $\mathcal{C}_{\text{est}}$ because both $R(v) + |\Delta(v)|$ and $|\{u \in T_v \mid r_{u,p(u)}.\texttt{est} \neq 0\}|$ are monotonically non-increasing once $r_{v,p(v)} \neq \bot$ holds in such an execution. Moreover, detailed analysis gives Lemma 15. whose proof are omitted due to the lack of space.

▶ **Lemma 15.** *Let $\Gamma = \gamma_0, \gamma_1, \ldots$ be an execution starting from $\mathcal{C}_{\text{est}}$. Then, $f(v_{\text{root}})$ decrements at least by one in eight rounds of Γ as long as $f(v_{\text{root}}) > 0$.*

▶ **Lemma 16.** *Every execution Γ of Base ends within $O(n\ell)$ rounds.*

Proof. Corollary 12, Lemma 14 and Lemma 15 imply that Γ reaches a configuration where $r_{v,p(v)}.\texttt{est} = 0$ and $\Delta(v) = 0$ for all $v \in V \setminus \{v_{\text{root}}\}$ within $O(n\ell)$ rounds. Pushing tokens to or pulling tokens from the external store, after an additional $O(\ell)$ rounds $\mathit{Diff}(v_{\text{root}}) = 0$. ◀

▶ **Theorem 17.** *Algorithm Base is a silent and self-stabilizing token distribution algorithm, which uses no work space per process and only constant work space per register, converges in $O(n\ell)$ rounds, and causes $\Theta(nh\epsilon)$ redundant token moves.*

4.2 Algorithm *SyncTokenDist*

Due to the lack of space, we present only an outline of the algorithm *SyncTokenDist* which reduces the number of redundant token moves of *Base*. The key idea is simple. Corollary 12 and Lemma 13 guarantee that every execution of *Base* enters a configuration in $\mathcal{C}_{\mathrm{est}}$ within $2h$ rounds and no redundant token moves happen thereafter. However, some processes can send or receive many tokens in the first $2h$ rounds, which makes $\Omega(nh\epsilon)$ redundant token moves in total in the worst case (Lemma 7). Algorithm *SyncTokenDist* simulates an execution of *Base* with a simplified version of the $\mathbb{Z}_3$ synchronizer [5], which loosely synchronizes an execution of *Base* so that the following property holds.

> For any integer x, if a process executes the procedure of *Base* at least $x+2$ times, then every neighboring process of the process must execute the procedure of *Base* at least x times.

This property and Lemma 10 guarantee that every process v can execute the procedure of *Base* at most $O(h)$ times until a configuration in $\mathcal{C}_{\mathrm{est}}$ is reached, after which no redundant token moves happen.

Specifically, in *SyncTokenDist*, every process v has variable $v.\mathtt{clock} \in \{0,1,2\}$ and $r_{v,u}.\mathtt{clock} \in \{0,1,2\}$ for every $u \in N(v)$. It always copies the latest value of $v.\mathtt{clock}$ to $r_{v,u}.\mathtt{clock}$ for all $u \in N(v)$. It also has all the variables of *Base*. We say that a process u is ahead of v if $u.\mathtt{clock} = v.\mathtt{clock}+1 \pmod 3$. Process v increments its clock when it is ahead of no neighbors and it has a neighbor that is ahead of v or when v is enabled to execute the procedure of *Base* and $v.\mathtt{clock} = u.\mathtt{clock}$ holds for all neighbors $u \in N(v)$. Process v executes the procedure of *Base* every time v increments its clock. It is easy to see that this simple algorithm satisfies the above property, hence we obtain the following theorem.

▶ **Theorem 18.** *Algorithm SyncTokenDist is a silent and self-stabilizing token distribution algorithm, which uses only constant work space per process and per register, converges in $O(n\ell)$ rounds, and causes $O(nh)$ redundant token moves.*

4.3 Algorithm *PIFTokenDist*

In this section, we present *PIFTokenDist*, which reduces the number of redundant token moves of *Base* from $O(nh\epsilon)$ to $O(n)$ but increases the convergence time from $O(n\ell)$ to $O(nh\ell)$. Algorithm *PIFTokenDist* uses Propagation of Information with Feedback (PIF) scheme [3] to reduce the number of redundant token moves. For our purpose, we use a simplified version of PIF. The pseudo code is shown in Algorithm 2. Each process v has a local variable $v.\mathtt{wave} \in \{0,1,2\}$, a shared variable $r_{v,u}.\mathtt{wave} \in \{0,1,2\}$ for all $u \in N(v)$, and all the variables of *Base*. Process v always copies the latest value of $v.\mathtt{wave}$ to $r_{v,u}.\mathtt{wave}$ for all $u \in N(v)$ (Line 4). An execution of *PIFTokenDist* repeats the cycle of three waves – the 0-wave, the 1-wave, and the 2-wave (Figure 5). Once $v_{\mathrm{root}}.\mathtt{wave} = 0$, the zero value is propagated from v_{root} to leaves (Line 1, the 0-value). In parallel, each process v changes $v.\mathtt{wave}$ from 0 to 1 after verifying that all its children already have the zero value in variable `wave` (Line 2, the 1-wave). When the 1-wave reaches a leaf, the wave bounces back to the root, changing the wave-value of processes from 1 to 2 (Line 3, the 2-wave). When the 2-wave reaches the root, it resets $v_{\mathrm{root}}.\mathtt{wave}$ to 0, thus the next cycle begins. A process v executes

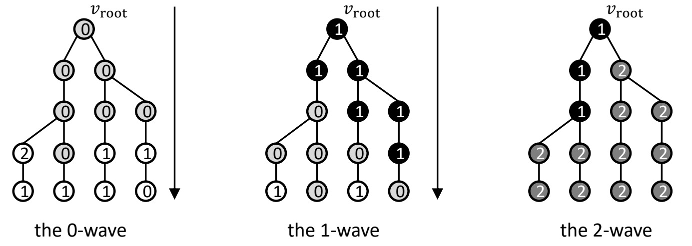


Figure 5 PIF waves of *PIFTokenDist*. A number in a circle represents the value of variable `wave`. Note that the 0-wave and the 1-wave run simultaneously.

Algorithm 2 *PIFTokenDist*.

[Actions of process v]

1: $v.\texttt{wave} \leftarrow 0$ **if** $(v = v_{\text{root}} \land v.\texttt{wave} = 2) \lor (v \neq v_{\text{root}} \land r_{p(v),v}.\texttt{wave} = 0)$
2: $v.\texttt{wave} \leftarrow 1$ **if** $(v.\texttt{wave} = 0) \land (v = v_{\text{root}} \lor r_{p(v),v}.\texttt{wave} = 1) \land \forall u \in C(v) : r_{u,v}.\texttt{wave} = 0)$
3: $v.\texttt{wave} \leftarrow 2$ and execute the procedure of *Base* **if** $(v.\texttt{wave} = 1) \land (\forall u \in C(v) : r_{u,v}.\texttt{wave} = 2)$
4: $r_{v,u}.\texttt{wave} \leftarrow v.\texttt{wave}$ for all $u \in N(v)$

the procedure of *Base* every time it receives the 2-wave, that is, every time it changes $v.\texttt{wave}$ from 1 to 2 (Line 3).

It is easy to see that every execution of *PIFTokenDist* reaches a configuration from which the cycle of three waves is repeated forever. Furthermore, the above PIF mechanism guarantees that, for any path $v_0, v_1, \ldots, v_\rho$ such that v_0 is a leaf, each process v_i $(1 \leq i \leq \rho)$ receives the 2-wave at least once in the order of $v_0, v_1, \ldots, v_\rho$ before v_ρ receives the 2-wave twice. Therefore, it holds by Lemma 10 that an execution of *PIFTokenDist* reaches a configuration in $\mathcal{C}_{\text{est}}$ before some process executes the procedure of *Base* more than three times. Hence, the number of redundant token moves is $O(n)$ in total. However, the convergence time increases from $O(n\ell)$ to $O(nh\ell)$ because it takes $O(h)$ rounds between every two consecutive executions of the procedure of *Base* at each process in an execution of *PIFTokenDist*. *PIFTokenDist*, shown in Algorithm 2 is not silent, but it can made silent by slightly modifying the algorithm, such that the root begins the 0-wave at Line 1 only when it detects that the simulated algorithm (*Base*) is not terminated. This modification is easily implemented by using the enabled-signal-propagation technique presented in [4].

▶ **Theorem 19.** *PIFTokenDist is a silent and self-stabilizing token distribution algorithm, which uses constant work space per process and per register, converges in $O(nh\ell)$ rounds, and permits $O(n)$ redundant token moves.*

5 Conclusion

We have given self-stabilizing and silent distributed algorithms for token distribution for rooted tree networks. The base algorithm *Base* converges in $O(n\ell)$ asynchronous rounds and causes $O(nh\epsilon)$ redundant token moves. Algorithms *SyncTokenDist* and *PIFTokenDist* use a synchronizer and a PIF scheme, respectively. Algorithm *SyncTokenDist* reduces the number of redundant token moves to $O(nh)$ without increasing convergence time while *PIFTokenDist* reduces the number of redundant token moves to $O(n)$, but increases the convergence time

to $O(nh\ell)$ rounds. All of the three algorithms uses constant memory space for each process and each link register.

References

1. Anish Arora and Mohamed G. Gouda. Load balancing: An exercise in constrainted convergence. In *Proceedings of the 9th International Workshop on Distributed Algorithms*, pages 183–197, 1995.
2. Andrei Z. Broder, Alan M. Frieze, Eli Shamir, and Eli Upfal. Near-perfect token distribution. In *Proceedings of the 19th International Colloquium on Automata, Languages and Programming*, pages 308–317, 1992.
3. Alain Bui, Ajoy K Datta, Franck Petit, and Vincent Villain. Snap-stabilization and PIF in tree networks. *Distributed Computing*, 20(1):3–19, 2007.
4. Ajoy K Datta, Laurence L Larmore, Toshimitsu Masuzawa, and Yuichi Sudo. A self-stabilizing minimal k-grouping algorithm. In *Proceedings of the 18th International Conference on Distributed Computing and Networking*, pages 3:1–3:10. ACM, 2017.
5. Ajoy K. Datta, Lawrence L. Larmore, and Toshimitsu Masuzawa. Constant space self-stabilizing center finding in anonymous tree networks. In *Proceedings of the International Conference on Distributed Computing and Networking*, pages 38:1–38:10, 2015.
6. EW Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of Computing Machinery*, 17:643–644, 1974.
7. Bhaskar Ghosh, Frank Thomson Leighton, Bruce M. Maggs, S. Muthukrishnan, C. Greg Plaxton, Rajmohan Rajaraman, Andréa W. Richa, Robert Endre Tarjan, and David Zuckerman. Tight analyses of two local load balancing algorithms. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, pages 548–558, 1995.
8. Kieran T. Herley. A note on the token distribution problem. *Inf. Process. Lett.*, 38(6):329–334, 1991.
9. Michael E. Houle, Antonios Symvonis, and David R. Wood. Dimension-exchange algorithms for load balancing on trees. In *Proceedings of the 9th International Colloquium on Structural Information and Communication Complexity*, pages 181–196, 2002.
10. Michael E. Houle, Ewan D. Tempero, and Gavin Turner. Optimal dimension-exchange token distribution on complete binary trees. *Theor. Comput. Sci.*, 220(2):363–376, 1999.
11. Michael E. Houle and Gavin Turner. Dimension-exchange token distribution on the mesh and the torus. *Parallel Computing*, 24(2):247–265, 1998.
12. Luciano Margara, Alessandro Pistocchi, and Marco Vassura. Perfect token distribution on trees. In *Proceedings of Structural Information and Communication Complexity, 11th International Colloquium*, pages 221–232, 2004.
13. D. Peleg. *Distributed computing: a locality-sensitive approach*, volume 5. Society for Industrial Mathematics, 2000.
14. David Peleg and Eli Upfal. The generalized packet routing problem. *Theor. Comput. Sci.*, 53:281–293, 1987.
15. David Peleg and Eli Upfal. The token distribution problem. *SIAM J. Comput.*, 18(2):229–243, 1989.
16. Yuichi Sudo, Ajoy K. Datta, Laurence L. Larmore, and Toshimitsu Masuzawa. Constant-space self-stabilizing token distribution in trees. In *Proceedings of 25th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, pages 25–29, 2018.