23rd International Conference on Principles of Distributed Systems

OPODIS 2019, December 17–19, 2019, Neuchâtel, Switzerland

Edited by

Pascal Felber
Roy Friedman
Seth Gilbert
Avery Miller

LIPIcs – Vol. 153 – OPODIS 2019 www.dagstuhl.de/lipics

Editors

Pascal Felber
University of Neuchâtel, Switzerland
pascal.felber@unine.ch

Roy Friedman
Technion, Israel
roy@cs.technion.ac.il

Seth Gilbert
NUS, Singapore
seth.gilbert@comp.nus.edu.sg

Avery Miller
University of Manitoba, Canada
avery.miller@umanitoba.ca

ACM Classification 2012
Computer systems organization → Dependable and fault-tolerant systems and networks; Computing methodologies → Distributed algorithms; Networks → Mobile networks; Networks → Wireless access networks; 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 978-3-95977-133-7

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-133-7.

Publication date
February, 2020

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

License
This work is licensed under a Creative Commons Attribution 3.0 Unported license (CC-BY 3.0):
https://creativecommons.org/licenses/by/3.0/legalcode.
In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:
- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPIcs.OPODIS.2019.0
ISBN 978-3-95977-133-7 **ISSN 1868-8969** **https://www.dagstuhl.de/lipics**

LIPIcs – Leibniz International Proceedings in Informatics

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

Editorial Board

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

ISSN 1868-8969

https://www.dagstuhl.de/lipics

Contents

Preface
Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller 0:ix–0:x

Program Committee
..... 0:xi

Steering Committee
..... 0:xiii

Organization Committee
..... 0:xv

List Of Authors
..... 0:xvii–0:xxi

Keynote Abstracts

Demystifying Bitcoin
Rachid Guerraoui 1:1–1:1

Distributed Optimization And Approximation: How Difficult Can It Be?
Keren Censor-Hillel 2:1–2:1

Snap ML – Accelerated Machine Learning for Big Data
Haris Pozidis 3:1–3:1

Regular Papers

FairLedger: A Fair Blockchain Protocol for Financial Institutions
Kfir Lev-Ari, Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi 4:1–4:17

Deconstructing Stellar Consensus
Álvaro García-Pérez and Maria A. Schett 5:1–5:16

Byzantine-Tolerant Set-Constrained Delivery Broadcast
Alex Auvolat, Michel Raynal, and François Taïani 6:1–6:23

Asymmetric Distributed Trust
Christian Cachin and Björn Tackmann 7:1–7:16

Uniform Partition in Population Protocol Model Under Weak Fairness
Hiroto Yasumi, Fukuhito Ooshita, and Michiko Inoue 8:1–8:16

Split and Migrate: Resource-Driven Placement and Discovery of Microservices at the Edge
Genc Tato, Marin Bertier, Etienne Rivière, and Cédric Tedeschi 9:1–9:16

HaTS: Hardware-Assisted Transaction Scheduler
Zhanhao Chen, Ahmed Hassan, Masoomeh Javidi Kishi, Jacob Nelson, and Roberto Palmieri 10:1–10:16

23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Minha: Large-Scale Distributed Systems Testing Made Practical
Nuno Machado, Francisco Maia, Francisco Neves, Fábio Coelho, and José Pereira 11:1–11:17

Fast Lean Erasure-Coded Atomic Memory Object
Kishori M. Konwar, N. Prakash , Muriel Médard, and Nancy Lynch 12:1–12:17

Interactive Coding Resilient to an Unknown Number of Erasures
Ran Gelles and Siddharth Iyer 13:1–13:16

A Generic Undo Support for State-Based CRDTs
Weihai Yu, Victorien Elvinger, and Claudia-Lavinia Ignat 14:1–14:17

In Search of the Fastest Concurrent Union-Find Algorithm
Dan Alistarh, Alexander Fedorov, and Nikita Koval 15:1–15:16

On Deterministic Linearizable Set Agreement Objects
Felipe de Azevedo Piovezan, Vassos Hadzilacos, and Sam Toueg 16:1–16:15

A Characterization of Consensus Solvability for Closed Message Adversaries
Kyrill Winkler, Ulrich Schmid, and Yoram Moses 17:1–17:16

An Efficient Universal Construction for Large Objects
Panagiota Fatourou, Nikolaos D. Kallimanis, and Eleni Kanellou 18:1–18:15

Toward Linearizability Testing for Multi-Word Persistent Synchronization Primitives
Diego Cepeda, Sakib Chowdhury, Nan Li, Raphael Lopez, Xinzhe Wang, and Wojciech Golab 19:1–19:17

Consensus in Equilibrium: Can One Against All Decide Fairly?
Itay Harel, Amit Jacob-Fanani, Moshe Sulamy, and Yehuda Afek 20:1–20:17

The Evolutionary Price of Anarchy: Locally Bounded Agents in a Dynamic Virus Game
Laura Schmid, Krishnendu Chatterjee, and Stefan Schmid 21:1–21:16

Tight Bounds on Distributed Exploration of Temporal Graphs
Tsuyoshi Gotoh, Paola Flocchini, Toshimitsu Masuzawa, and Nicola Santoro 22:1–22:16

Parallel and Distributed Algorithms for the Housing Allocation Problem
Xiong Zheng and Vijay K. Garg 23:1–23:16

Oblivious Permutations on the Plane
Shantanu Das, Giuseppe A. Di Luna, Paola Flocchini, Nicola Santoro, Giovanni Viglietta, and Masafumi Yamashita 24:1–24:16

On Memory, Communication, and Synchronous Schedulers When Moving and Computing
Paola Flocchini, Nicola Santoro, and Koichi Wada 25:1–25:17

Lower Bounds for Shoreline Searching With 2 or More Robots
Sumi Acharjee, Konstantinos Georgiou, Somnath Kundu, and Akshaya Srinivasan 26:1–26:11

Gathering on Rings for Myopic Asynchronous Robots With Lights
Sayaka Kamei, Anissa Lamani, Fukuhito Ooshita, Sébastien Tixeuil, and Koichi Wada 27:1–27:17

Optimal Register Construction in M&M Systems
Vassos Hadzilacos, Xing Hu, and Sam Toueg 28:1–28:16

Linearizable Replicated State Machines With Lattice Agreement
Xiong Zheng, Vijay K. Garg, and John Kaippallimalil 29:1–29:16

Exact Byzantine Consensus on Arbitrary Directed Graphs Under Local Broadcast Model
Muhammad Samir Khan, Lewis Tseng, and Nitin H. Vaidya 30:1–30:16

Reconfigurable Lattice Agreement and Applications
Petr Kuznetsov, Thibault Rieutord, and Sara Tucci-Piergiovanni 31:1–31:17

Towards Distributed Two-Stage Stochastic Optimization
Yuval Emek, Noga Harlev, and Taisuke Izumi 32:1–32:16

Equivalence Classes and Conditional Hardness in Massively Parallel Computations
Danupon Nanongkai and Michele Scquizzato 33:1–33:16

Sparse Hopsets in Congested Clique
Yasamin Nazari 34:1–34:16

Massively Parallel Approximate Distance Sketches
Michael Dinitz and Yasamin Nazari 35:1–35:17

Preface

The papers in this volume were presented at the 23rd International Conference on Principles of Distributed Systems (OPODIS 2019), held on December 17–19, 2019, in Neuchâtel, Switzerland. The conference was organized by the University of Neuchâtel and took place at the Faculty of Sciences.

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
- Cloud computing and data centers
- Dependable distributed algorithms and systems
- Design and analysis of concurrent and distributed data structures
- Design and analysis of distributed graph algorithms
- Distributed deployments of machine learning
- Distributed event processing
- Distributed operating systems, middleware, and distributed database systems
- Distributed storage and file systems, large-scale systems, and big data analytics
- Edge computing
- Embedded and energy-efficient distributed systems
- Game-theory and economical aspects of distributed computing
- High-performance, cluster, cloud and grid computing
- Impossibility results for distributed computing
- Internet of things and cyber-physical systems
- Mesh and ad-hoc networks networks
- Mobile agents, robots, and rendezvous
- Programming languages, formal methods, specification and verification
- Randomization in distributed computing
- Security and privacy, cryptographic protocols
- Self-stabilization, self-organization, autonomy
- Shared memory algorithms
- Social systems, peer-to-peer and overlay networks
- Specification and verification of distributed systems
- Synchronization
- Transactional memory

We received 87 submissions, each of which underwent a double-blind peer review 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 87 submissions, 32 papers were selected to be included in these proceedings.

23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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.

Two Best Paper Awards were given: one award was given to Nikita Koval, Dan Alistarh, and Alexander Federov for their paper titled "In Search of the Fastest Concurrent Union-Find Algorithm", and the other award was given to Kfir Lev-Ari, Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi for their paper titled "FairLedger: A Fair Blockchain Protocol for Financial Institutions". The Best Student Paper Award was given to Yasamin Nazari for their paper titled "Massively Parallel Approximate Distance Sketches", co-authored with Michael Dinitz.

This year OPODIS had three distinguished invited keynote speakers: Haris Pozidis (IBM Research), Keren Censor-Hillel (Technion), and Rachid Guerraoui (EPFL).

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

Organizing this event would not have been possible without the time and the effort of the Organizing Committee, notably: Valerio Schiavoni, who was responsible for local arrangements and the website; Peter Kropf, who managed sponsorships; and Avery Miller, who managed the proceedings.

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

December 2019

Pascal Felber (University of Neuchâtel, Switzerland)
Roy Friedman (Technion, Israel)
Seth Gilbert (NUS, Singapore)

Program Committee

General Chair

Pascal Felber, University of Neuchâtel, Switzerland

Program Chairs

Roy Friedman, Technion, Israel

Seth Gilbert, National University of Singapore, Singapore

Program Committee

James Aspnes, Yale University, USA

John Augustine, Indian Institute Of Technology Madras, India

Michael A. Bender, Stony Brook University, USA

Natacha Crooks, The University of Texas at Austin, USA

Gianlorenzo D'Angelo, Gran Sasso Science Institute - GSSI, Italy

Xavier Défago, Tokyo Institute of Technology, Japan

Elias Duarte, Federal University of Parana, Brazil

Gil Einziger, Technion, Israel

Paola Flocchini, University of Ottawa, Canada

Pierre Fraigniaud, CNRS and University Paris Diderot, France

Cyril Gavoille, LaBRI, University of Bordeaux, France

Valerie Issarny, INRIA, France

Alex Kogan, Oracle Labs, USA

Fabian Kuhn, University of Freiburg, Germany

Alberto Montresor, University of Trento, Italy

Adam Morrison, Tel Aviv University, Israel

Calvin Newport, Georgetown University, USA

Gopal Pandurangan, University of Houston, USA

Marta Patino, Universidad Politécnica de Madrid, Spain

Sebastiano Peluso, Amazon Web Services, USA

Jose Pereira, University of Minho, Portugal

Peter Robinson, McMasters University, Canada

Luis Rodrigues, Universidade de Lisboa, Portugal

Christian Scheideler, University of Paderborn, Germany

Roman Vitenberg, University of Oslo, Norway

Chaodong Zheng, Nanjing University, China

23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Steering Committee

Sébastien Tixeuil, Sorbonne University, France (chair)

James Aspnes, Yale University, USA

Xavier Défago, Tokyo Institute of Technology, Japan

Faith Ellen, University of Toronto, Canada

Pascal Felber, University of Neuchâtel, Switzerland

Alessia Milani, University of Bordeaux, France

Luis Rodrigues, University of Lisboa, Portugal

Yukiko Yamauchi, University Kyushu, Japan

23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Organization Committee

Valerio Schiavoni (Local Organization Chair), University of Neuchâtel, Switzerland

Peter Kropf (Finance Chair), University of Neuchâtel, Switzerland

Avery Miller (Proceedings Chair), University of Manitoba, Canada

23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

List of Authors

Sumi Acharjee
Ryerson University
Canada
sumi.acharjee@ryerson.ca

Yehuda Afek
Tel-Aviv University
Israel
yehuda.afek@gmail.com

Dan Alistarh
IST Austria
Austria
d.alistarh@gmail.com

Alex Auvolat
Univ Rennes, Inria, CNRS, IRISA
France
alex.auvolat@inria.fr

Marin Bertier
INSA Rennes, Univ Rennes, Inria, CNRS, IRISA, France
France
Marin.Bertier@irisa.fr

Christian Cachin
University of Bern
Switzerland
cachin@acm.org

Diego Cepeda
University of Waterloo
Canada
dcepeda@uwaterloo.ca

Krishnendu Chatterjee
Institute of Science and Technology (IST)
Austria
krish.chat@gmail.com

Zhanhao Chen
Lehigh University
United States
zhc416@lehigh.edu

Sakib Chowdhury
University of Waterloo
Canada
sakib.chowdhury@uwaterloo.ca

Fábio André Coelho
INESCTEC
Portugal
facoelho@inesctec.pt

Shantanu Das
Aix-Marseille University
France
shantanu.das@lis-lab.fr

Giuseppe Antonio Di Luna
DIAG, University of Rome “Sapienza”
Italy
g.a.diluna@gmail.com

Michael Dinitz
Johns Hopkins University
United States
mdinitz@cs.jhu.edu

Victorien Elvinger
Université de Lorraine
France
victorien.elvinger@loria.fr

Yuval Emek
Technion
Israel
yuval.emek@gmail.com

Panagiota Fatourou
FORTH ICS and University of Crete
Greece
faturu@csd.uoc.gr

Alexander Fedorov
JetBrains, HSE
Russia
aleksandr.fyodorov@jetbrains.com

Paola Flocchini
University of Ottawa
Canada
flocchin@site.uottawa.ca

Álvaro García-Pérez
IMDEA Software Institute
Spain
alvaro.garcia.perez@imdea.org

23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Vijay Garg
The University of Texas at Austin
United States
garg@ece.utexas.edu

Ran Gelles
Bar-Ilan University
Israel
ran.gelles@biu.ac.il

Konstantinos Georgiou
Ryerson University
Canada
konstantinos@ryerson.ca

Wojciech Golab
University of Waterloo
Canada
wgolab@uwaterloo.ca

Tsuyoshi Gotoh
Osaka University
Japan
t-gotoh@ist.osaka-u.ac.jp

Vassos Hadzilacos
University of Toronto
Canada
vassos@cs.toronto.edu

Itay Harel
Tel-Aviv University
Israel
itayelf@gmail.com

Noga Harlev
Technion
Israel
snogazur@campus.technion.ac.il

Ahmed Hassan
Virginia Tech
United States
ahmed.hassan@alexu.edu.eg

Xing Hu
University of Toronto
Canada
xing.hu.utoronto@gmail.com

Claudia-Lavinia Ignat
INRIA
France
ignatcla@loria.fr

Michiko Inoue
Nara Institute of Science and Technology
Japan
kounoe@is.naist.jp

Siddharth Iyer
University of Washington
United States
sviyer97@gmail.com

Taisuke Izumi
Nagoya Institute of Technology
Japan
t-izumi@nitech.ac.jp

Amit Jacob-Fanani
Tel-Aviv University
Israel
amitjf@gmail.com

John Kaippallimalil
Huawei
United States
John.Kaippallimalil@huawei.com

Nikolaos D. Kallimanis
FORTH ICS
Greece
nkallima@ics.forth.gr

Sayaka Kamei
Department of Information Engineering,
Graduate School of Engineering, Hiroshima
University
Japan
s-kamei@se.hiroshima-u.ac.jp

Eleni Kanellou
FORTH ICS
Greece
kanellou@ics.forth.gr

Idit Keidar
Technion - Israel Institute of Technology
Israel
idish@ee.technion.ac.il

Muhammad Khan
University of Illinois at Urbana-Champaign
United States
mskhan6@illinois.edu

Masoomeh Javidi Kishi
Lehigh University
United States
maj717@lehigh.edu

Kishori Konwar
RLE, MIT
United States
kishori@csail.mit.edu

Nikita Koval
IST Austria, JetBrains
Austria
ndkoval@ya.ru

Somnath Kundu
Ryerson University
Canada
somnath.kundu@ryerson.ca

Petr Kuznetsov
Telecom Paris, Institut Polytechnique Paris
France
petr.kuznetsov@telecom-paristech.fr

Anissa Lamani
EISTI
France
anissa.lamani@gmail.com

Kfir Lev-Ari
Technion - Israel Institute of Technology
Israel
kfirlevari@gmail.com

Nan Li
University of Waterloo
Canada
nan.li@uwaterloo.ca

Raphael Lopez
University of Waterloo
Canada
rwlopez@edu.uwaterloo.ca

Nancy Lynch
Massachusetts Institute of Technology
United States
lynch@csail.mit.edu

Nuno Machado
INESCTEC
Portugal
nuno.a.machado@inesctec.pt

Francisco Maia
INESCTEC
Portugal
francisco.a.maia@inesctec.pt

Dahlia Malkhi
Calibra
United States
dahliamalkhi@gmail.com

Toshimitsu Masuzawa
Osaka University
Japan
masuzawa@ist.osaka-u.ac.jp

Yoram Moses
Technion
Israel
moses@ee.technion.ac.il

Medard Muriel
RLE, MIT
United States
medard@mit.edu

Danupon Nanongkai
KTH Royal Institute of Technology
Sweden
danupon@gmail.com

Yasamin Nazari
Johns Hopkins University
United States
ynazari@jhu.edu

Jacob Nelson
Lehigh University
United States
jjn217@lehigh.edu

Francisco Neves
INESCTEC & U. Minho
Portugal
fntneves@gmail.com

Fukuhito Ooshita
NAIST
Japan
f-oosita@is.naist.jp

Roberto Palmieri
Computer Science and Engineering, Lehigh University
United States
palmieri@lehigh.edu

Jose Pereira
INESCTEC & U. Minho
Portugal
jop@di.uminho.pt

Felipe de Azevedo Piovezan
University of Toronto
Canada
felipe@cs.toronto.edu

N Prakash
MIT
United States
prakashnarayanamoorthy@gmail.com

Michel Raynal
Univ Rennes, Inria, CNRS, IRISA
France
michel.raynal@irisa.fr

Thibault Rieutord
CEA
France
thibault.rieutord@cea.fr

Etienne Rivière
UCLouvain
Belgium
etienne.riviere@uclouvain.be

Nicola Santoro
Carleton University
Canada
santoro@scs.carleton.ca

Maria A. Schett
University College London
United Kingdom
mail@maria-a-schett.net

Laura Schmid
IST Austria
Austria
laura.schmid@ist.ac.at

Stefan Schmid
University of Vienna, Austria
Austria
schmiste@gmail.com

Ulrich Schmid
Vienna University of Technology
Austria
s@ecs.tuwien.ac.at

Michele Scquizzato
University of Padova
Italy
scquizza@dei.unipd.it

Alexander Spiegelman
VMware Research
Israel
sasha.spiegelman@gmail.com

Akshaya Srinivasan
National Institute of Technology, Tiruchirappalli
India
akshaya.kms@gmail.com

Moshe Sulamy
Tel-Aviv University
Israel
moshesulamy@mail.tau.ac.il

Björn Tackmann
DFINITY
Switzerland
bjoern.tackmann@ieee.org

François Taïani
Univ Rennes, Inria, CNRS, IRISA
France
francois.taiani@irisa.fr

Genc Tato
Univ Rennes, Inria, CNRS, IRISA, France
France
genc.tato@irisa.fr

Cédric Tedeschi
Univ Rennes, Inria, CNRS, IRISA, France
France
cedric.tedeschi@inria.fr

Sebastien Tixeuil
Universite Pierre et Marie Curie
France
Sebastien.Tixeuil@lip6.fr

Sam Toueg
University of Toronto
Canada
sam@cs.toronto.edu

Lewis Tseng
Boston College
United States
lewis.tseng@bc.edu

Sara Tucci
CEA
France
sara.tucci@cea.fr

Nitin Vaidya
Georgetown University
United States
nitin.vaidya@georgetown.edu

Giovanni Viglietta
JAIST
Japan
viglietta@gmail.com

Koichi Wada
Hosei University
Japan
wada@hosei.ac.jp

Xinzhe Wang
University of Waterloo
Canada
x793wang@edu.uwaterloo.ca

Kyrill Winkler
Vienna University of Technology
Austria
kwinkler@ecs.tuwien.ac.at

Masafumi Yamashita
Kyushu University
Japan
masafumi.yamashita@gmail.com

Hiroto Yasumi
Nara Institute of Science and Technology
Japan
yasumi.hiroto.yf9@is.naist.jp

Weihai Yu
UIT The Arctic University of Norway
Norway
weihai@cs.uit.no

Xiong Zheng
The University of Texas at Austin
United States
zhengxiongtym@utexas.edu

Demystifying Bitcoin

Rachid Guerraoui
School of Computer and Communication Sciences, EPFL, Switzerland
https://lpdwww.epfl.ch/rachid/
rachid.guerraoui@epfl.ch

Abstract

This talk will explain the bitcoin algorithm from the distributed computing perspective, precisely define the underlying double-payment problem, and present a much simpler alternative to solve the problem without relying on consensus and consuming so much energy.

Rachid Guerraoui is professor in Computer Science at EPFL where he leads the Distributed Computing Laboratory. He worked in the past with École des Mines de Paris, CEA Saclay, HP Labs in Palo Alto and MIT. He has been elected ACM Fellow and Professor of the College de France. He was awarded a Senior ERC Grant and a Google Focused Award.

2012 ACM Subject Classification Information systems → Electronic commerce; Computing methodologies → Distributed algorithms

Keywords and phrases Bitcoin, Payment systems

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.1

Category Keynote Abstract

© Rachid Guerraoui;
licensed under Creative Commons License CC-BY
23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller; Article No. 1; pp. 1:1–1:1
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Distributed Optimization And Approximation: How Difficult Can It Be?

Keren Censor-Hillel
Technion, Haifa, Israel
http://ckeren.net.technion.ac.il
ckeren@cs.technion.ac.il

Abstract

We know that exact distributed algorithms for optimization problems cannot be fast. To overcome these barriers, very efficient approximation algorithms have been designed in various distributed settings. But for very small approximation factors, a mystery remains: Why do we not have fast distributed algorithms for very small approximations factors in bandwidth restricted settings?

This talk will overview the state-of-the-art of distributed optimization and approximation algorithms and discuss the major challenges in determining the complexity of small approximations.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Mathematics of computing → Approximation algorithms

Keywords and phrases Distributed graph algorithms, Optimization and approximations

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.2

Category Keynote Abstract

Funding Keren Censor-Hillel's research has received funding from the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme (grant agreement No 755839).

© Keren Censor-Hillel;
licensed under Creative Commons License CC-BY
23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller; Article No. 2; pp. 2:1–2:1
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Snap ML – Accelerated Machine Learning for Big Data

Haris Pozidis
IBM Research Zurich, Switzerland

Abstract

Snap Machine Learning (Snap ML) is a new software library for training popular machine learning models, characterized by very high performance, scalability to TB-scale datasets and high resource efficiency. It continuously evolves and currently supports generalized linear models, decision trees, random forests and gradient boosting machines. Snap ML has been built to address the needs of business applications, which often have to deal with high-volume data, react fast to changing environments, and use resources efficiently to drive down cost. The high efficiency of Snap ML, in particular in dealing with big data, comes from innovations in distributed optimization, among other things. This talk will review the principles of the Snap ML library, explain how it achieves high speed and scalability, and present several cases of business workloads that demonstrate the benefits offered by Snap ML.

Haris Pozidis manages the Cloud Storage and Analytics group at IBM Research in Zurich, Switzerland. He was with Philips Research, Eindhoven, The Netherlands, before joining IBM. He has worked on read channel design for DVD and Blu-ray Disc at Philips, and played a key role in developing the first scanning probe-based data storage system at IBM, the "Millipede". His current focus is on the development of Flash memory controllers for all-flash arrays, on phase change memory technology and system solutions, and on accelerated software libraries for machine learning. He holds over 120 US patents, has co-authored more than 120 publications, is an IBM Principal Research Scientist and Master Inventor, and a Senior Member of the IEEE.

2012 ACM Subject Classification Computing methodologies → Machine learning; Software and its engineering → Software libraries and repositories

Keywords and phrases Machine Learning, Big Data

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.3

Category Keynote Abstract

© Haris Pozidis;
licensed under Creative Commons License CC-BY
23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller; Article No. 3; pp. 3:1–3:1
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

FairLedger: A Fair Blockchain Protocol for Financial Institutions

Kfir Lev-Ari
Technion IIT, Haifa, Israel

Alexander Spiegelman[1]
VMware Research, Herzliya, Israel

Idit Keidar
Technion IIT, Haifa, Israel

Dahlia Malkhi
Calibra, Menlo Park, CA, USA

Abstract

Financial institutions nowadays are looking into technologies for permissioned blockchains. A major effort in this direction is Hyperledger, an open source project hosted by the Linux Foundation and backed by a consortium of over a hundred companies. A key component in permissioned blockchain protocols is a byzantine fault tolerant (BFT) consensus engine that orders transactions. However, currently available BFT solutions in Hyperledger (as well as in the literature at large) are inadequate for financial settings; they are not designed to ensure fairness or to tolerate the selfish behavior that inevitably arises when financial institutions strive to maximize their own profit.

We present FairLedger, a permissioned BFT blockchain protocol, which is fair, deigned to deal with rational behavior, and, no less important, easy to understand and implement. Our secret sauce is a new communication abstraction called *detectable all-to-all (DA2A)*, which allows us to detect players (byzantine or rational) that deviate from the protocol and punish them. We implement FairLedger in the Hyperledger open source project using the Iroha framework – one of the biggest projects therein. To evaluate FairLegder's performance, we also implement it in the PBFT framework and compare the two protocols. Our results show that in failure-free scenarios in wide-area settings, FairLedger achieves better throughput than both Iroha's implementation and PBFT.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Blockchain, Fairness, Byzantine fault tolerance, Rational players, Equilibrium

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.4

Related Version A full version of the paper is available at `https://arxiv.org/abs/1906.03819`.

1 Introduction

As of today, support for financial transactions between institutions is limited, slow, and costly. For example, an oversees money transfer between two banks might take several days and entail fees of tens of dollars. The source of this cost (in term of both time and money) is the need for a reliable clearing house; sometimes this even requires physical phone calls at the end of the day. At the same time, emerging decentralized cryptocurrencies like Bitcoin [42] complete transactions within less than hour, at a cost of microcents. It is therefore not surprising that financial institutions are looking into newer technologies to bring them up to speed and facilitate trading in today's global economy.

The most prominent technology considered in this context is that of a *blockchain*, which implements a secure peer-to-peer *ledger* of financial transactions on top of a consensus engine.

[1] Kfir Lev-Ari and Alexander Spiegelman contributed equally to this work.

© Kfir Lev-Ari, Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi;
licensed under Creative Commons License CC-BY
23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller; Article No. 4; pp. 4:1–4:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

A major effort in this direction is Hyperledger [28], an open-source project hosted by the Linux Foundation and backed by a consortium of more than a hundred companies. Unlike anonymous cryptocurrencies with open participation, in blockchains for financial institutions – also called *permissioned blockchains* – every participant is pre-known and certified, so that it has to be responsible for its actions in the real world. Permissioned blockchains [28, 40, 45] thus abandon the slow and energy-consuming proof-of-work paradigm of Bitcoin, and tend to go back to more traditional distributed consensus protocols. Because of the high stakes, malicious deviations from the protocol (due to bugs or attacks), rare as they might be, should never compromise the service. Such deviations are modeled as *byzantine* faults [34], and to deal with them, proposed solutions use *byzantine fault tolerant (BFT)* consensus protocols.

Yet we believe that dealing with byzantine failures is only a small part of what is required in permissioned blockchains. In fact, a break-in that causes a bank's software to behave maliciously is so unusual that it is a top news story and is investigated by the FBI. On the other hand, financial institutions always try to maximize their own profit, and would never use a system that discriminates against them. Moreover, they can be expected to selfishly deviate from the protocol whenever they can benefit from doing so. In particular, financial entities typically receive a fee for every transaction they append to the shared ledger, and can thus be expected to attempt to game the system in a way that maximizes the rate of their own transactions in the ledger. Such *rational* behavior, if not carefully considered, not only can discriminate against some entities, but may also compromise safety.

Thus, in the FinTec context, one faces a number of important challenges that were not always emphasized in previous BFT work: (1) *fairness* in terms of the opportunities each participant gets to append transactions to the ledger; (2) expected *rational behavior* by all players; and (3) *optimized failure-free performance* in wide-area setting, given that financial institutions are usually very secure and inter-institutional platforms would be deployed over a secure WAN. In addition, it is important to stress (4) *protocol simplicity*, because complex protocols are inherently bug-prone and easier to attack. In this work we develop *FairLedger*, a new permissioned BFT blockchain protocol for the Hyperledger framework, which addresses all of these challenges. Our protocol is fair, designed for rational participants, optimized for the failure-free case, simple to understand, and easy to implement. Specifically, we show that following the protocol is an equilibrium, and that when rational participants do follow the protocol, they all get perfectly fair shares of the ledger.

Given that byzantine failures are rare, our philosophy is to optimize for the *normal mode* when they do not occur (as also emphasized in some previous works, e.g., Zyzzyva [32]). For this mode, we design a simple protocol that provides high performance when all players are rational but not byzantine. Under byzantine failures, the normal mode protocol remains safe and fair, but may lose progress. Upon detecting that a rogue participant is attempting to prevent progress, we switch to the *alert mode*. At this point, it is expected that real-world authorities (such as the FBI or Interpol) will step in to investigate the break-in. But such an investigation may take days to complete, and in the time being, the service remains operational – albeit slower – using the alert mode protocol.

An important lesson learned from the deployment of Paxos-like protocols in real systems such as ZooKeeper [31] and etcd [19] is that systems will only be used if they are easy to understand, implement, and maintain. Like these systems, we follow the Vertical Paxos [4, 33] approach of using a fixed set of participants (sometimes called quorum) for sequencing transactions and reconfiguring this set upon failures. Specifically, we designate a *committee* consisting of all the participants who are interested in issuing transactions and have them run a *sequencing protocol* to order their transactions. A complementary *master* service monitors

the committee's progress and initiates reconfiguration when needed. Including all interested players in the committee is instrumental for fairness – this way, all committee members benefit from sequencing batches that include transactions by all of them.

We assume a loosely synchronous model, where a master can use a coarse time bound (e.g., one minute) to detect lack of progress. This bound is only used for failure recovery, and does not otherwise affect performance. A key feature of our alert mode is that whenever participants deviate from the protocol in a way that jeopardizes progress, they are accurately detected and so can be removed from the committee. Unlike in other Hyperledger protocols [45], FairLedger never indicts correct participants, allowing the system to heal itself following attacks.

The sequencing protocol uses all-to-all exchange of signed messages among committee members. Since the committee includes all participants and all messages are signed, the protocol can ensure safety despite byzantine failures of almost any minority. Specifically, for f failures, our protocol is correct whenever the number of participants satisfies $n \geq 2f + 3$. The flip side is that it is enough for one participant to withhold a single message in order to prevent progress. Such a deviation from the protocol is tricky to detect as one participant can claim that it had sent a message to another, while the recipient claims that the message has not arrived. To deal with such deviations, we define a new communication abstraction, which we call *detectable all-to-all (DA2A)*. Besides the standard *broadcast* and *deliver* API, DA2A exposes a *detect* method that returns an accurate and complete set of deviating participants.

We implement FairLedger's sequencing protocol in Iroha [45], which is part of the Hyperledger [28] open-source project, and compare its performance to their implementation. Specifically, since Iroha's implementation is modular, we are able to replace their BFT consensus protocol, (which is based on [23]), with our sequencing protocol without changing other components (e.g., communication, cryptographic, and database libraries). Experiments over WAN emulation [48] show that FairLeadger outperforms Iroha's BFT protocol in the vast majority of the tested scenarios (both in normal mode and in alert mode).

Since the Iroha system consists of many components (e.g., GRPC [30] communication) that may induce overhead, we also implement FairLedger's sequencing protocol in the PBFT [17] framework, which provides a clean environment to evaluate raw consensus performance. Our results show that Fairledger's latency is better than PBFT's in both the normal and alert modes. Fairledger's throughput exceeds PBFT's in normal mode but is inferior to it in the alert mode, although PBFT's advantage diminishes as the system scale grows.

In summary, this paper makes the following contributions:

1. We define a fair distributed ledger abstraction for rational participants.
2. We define a detectable all-to-all (DA2A) abstraction as a building block for such ledgers.
3. We design FairLedger, the first BFT blockchain protocol that ensures strong fairness when all participants are rational. FairLedger is safe under byzantine failures of almost any minority, and detects and punishes deviating (byzantine and rational) participants. It is also simple to understand and implement.
4. We substitute Iroha, which is one of the Hyperledger's existing sequencing protocol, with FairLedger with improved performance. We also implement FairLedger's sequencing protocol in the PBFT framework; FairLedger outperforms PBFT in the normal mode but achieves slightly lower throughput in the alert mode.

2 Problem Definition and System Model

We consider a set of players, each representing a real-world financial entity, jointly attempting to agree on a shared *ledger* of financial transactions. Every player has an unbounded stream of transactions that it wants to append to the ledger and we assume that the player benefits from doing so. A principal goal for our service is *fairness*, that is, providing all entities with equal opportunities for appending transactions.

2.1 Byzantine and rational behavior

Traditional distributed systems are managed by a single organization, where deviation from the protocol – referred to as byzantine behavior – is explained as a bug or by the deviating entity being hacked, and only a small subset of the players are byzantine. In this work, however, we seek a protocol that coordinates among many organizations that trade with financial assets. We thus have to take into account that *every* entity may behave *rationally*, and deviate from the protocol if doing so increases its benefit.

To reason about such rational behavior we assume that each entity can be either *byzantine* or *rational* [5, 36, 41]. A rational entity has a known utility function that it tries to maximize and deviates from the protocol only if this increases its utility, whereas a byzantine entity can deviate arbitrarily from the protocol (e.g., crash, withhold messages, or send incorrect protocol messages), i.e., its utility function is unknown.

Our system involves two types of entities – *players* and *auditors*. Players (e.g., banks) propose transactions to append to the ledger, while auditors oversee the system. The same physical entity may be both a player and an auditor, but other entities (e.g., government central banks) may also act as auditors. There are initially n players and any number of auditors. The number of byzantine players is bounded by a known parameter f, where $n \geq 2f + 3$. At most a minority of the auditors can be byzantine.

In order to prove that a protocol is correct in our model, we need to show that following the protocol is an equilibrium for rational entities even in the presence of f byzantine faults.

2.2 Distributed fair ledger

A *ledger* is an abstract object that maintains a log (i.e., sequence) of *transactions* from some domain $\mathcal{T}$. It supports two operations with the following sequential specification: An *append*(t), $t \in \mathcal{T}$, changes the state of the log by appending t to its end. A *read*(l) operation returns the last l transactions in the log. The log is initially empty.

The *utility function of a rational player* is the ratio of transactions that it appends to the ledger, i.e., the number of transactions it appends to the ledger out of the total number of transactions in the ledger. Between two ledgers with the same ratio, the longer one is preferred. This models players who care about the overall system progress but care more about getting their fair share of it.

The *utility function of an auditor* is the committee size in case progress is being made, and 0 in case the system stalls. In other words, the auditors aim to ensure the system's overall health. In case an entity acts as an auditor and as a player, the auditor's utility is the dominating and the player's utility breaks ties.

We require '*strict fairness*. Intuitively, this means that for every player p_1 that follows the protocol, at any point when the log contains k transactions appended by p_1, the log does not contain more than $k + 1$ transactions appended by any other player. In the full paper [35] we formalize and extend this definition to allow differential quality of service, whereby different players are allocated different shares of the log and these shares may change over time.

2.3 System model

We assume that players have been certified by some trusted certification authority known to all players. In addition, we assume a PKI [44]: each player has a unique pair of public and private cryptographic keys, where the public keys are known to all players, and the adversary does not have enough computational power to unravel non-byzantine players' private keys.

We assume reliable communication channels between pairs of players. As in previous works on permissioned blockchains [23, 28, 45], we assume that there is a known upper bound Δ on message latency. Nevertheless, our sequencing protocol is safe and fair even if the bound does not hold. We exploit this bound to detect failures when the protocol stalls because a rogue player deviates from the protocol by withholding messages. Thus, the bound can be set very conservatively (e.g., in the order of minutes) so as to avoid false detection.

3 Solution Components

Our goal is to design a ledger that financial institutions will be able to use. Such a protocol, besides being fair, secure against malicious attacks, and resilient to selfish behavior, must be simple to understand, implement, and maintain. Therefore, although we appreciate complex protocols with many corner cases and clever optimizations, we try here to keep the design as simple as possible. The simple design not only reduces vulnerabilities, it also makes it much easier to reason about selfish behavior.

Committee and master. We adopt the Vertical Paxos [4, 33] paradigm, where a single committee (known to all) partakes in agreeing on all transactions. Initially, the committee consists of all players. By requiring all committee members to endorse transactions, we create an incentive for all of them to append to the log batches including transactions from all of them. To handle cases when committee members stop responding (e.g., due to a crash or an attack), a complementary *master* service performs *reconfiguration*: detecting such members and removing (or replacing) them. Thus, we logically implement two components: (1) a committee that runs the sequencing protocol and (2) a master responsible for progress. The master is implemented by auditors using a minority-resilient synchronous BFT protocol like [21]; its impact on overall system performance is small, and so we do not optimize its implementation. For the remainder of this paper, we abstract away this protocol and simply treat the master as a single trusted authority.

Detection of misbehavior. The master's ability to evict deviating (byzantine or rational) players relies on its ability to detect deviations from the protocol. We divide the possible deviations into two categories: *active* and *passive*. An active deviation occurs when a player sends messages that do not coincide with the protocol. By singing all messages with private keys, we achieve non-repudiation, i.e., messages can be linked to their senders and provide evidence of misbehavior, which the master can use to detect deviation.

Passive deviation, which stalls the protocol by withholding messages, is much harder to detect. For example, if the protocol hangs waiting for p_1 to take an action following a message it expects from p_2, we cannot, in general, know if p_2 is the culprit (because it never sent a message to p_1) or p_1 is at fault.

To address this challenge we present our novel DA2A broadcast abstraction, which supports *broadcast*(m) and *deliver(m)* operations for the players and a *detect()* operation for the master. Every player p_i invokes *broadcast*(m) for some message m s.t. all the other

players should *deliver*(m). The *detect()* operation performed by the master returns a set S of players that deviate from the protocol together with corresponding proofs:

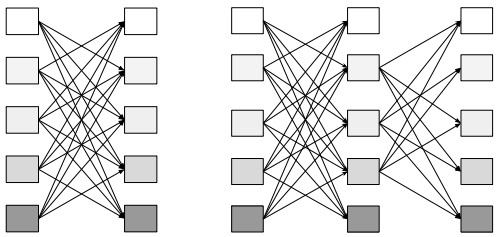
▶ **Definition 1** (Detectability). *For every two players p_j, p_i s.t. p_i does not deliver a message from p_j, S contains p_j (with a proof of p_j's deviation) in case p_j did not perform broadcast(m) properly, and otherwise, it contains p_i (with a proof of p_i's deviation). Moreover, S contains only deviating players.*

Note that in case S is empty, all the players follow the protocol, meaning that all the players broadcast a message and deliver messages broadcast by all other players.

4 FairLedger Protocol

We present our detectable all-to-all building block in Section 4.1, then use it for our sequencing protocol in Section 4.2, and for the recovery protocol in Section 4.3. In Section 4.4, we informally argue that following the protocol is a Nash equilibrium. For space limitations, the full correctness proof (including game theoretical analysis) is deferred to the full paper [35].

4.1 Detectable all-to-all (DA2A)



(a) direct all-to-all. **(b)** relayed all-to-all.

Figure 1 All-to-all communication patterns.

Communication patterns. We start by discussing two ways to implement all-to-all communication over reliable links. The simplest way to do so is *direct all-to-all*, in which *broadcast(m)* sends message m to all other players (see Figure 1a). This implementation has the optimal cost of 1 hop and $n(n-1)$ messages, but cannot reveal any information about passive deviations: In case p_i does not deliver a message from p_j, the master has no way of knowing whether p_j did not send a message to p_i, or p_i is lying about not receiving the message.

Another approach, which we call *relayed all-to-all*, designates a subset of the players as *relays*. A *broadcast(m)* sends m to all players, and when a relay receives a message for the first time, it forwards it to all players (see Figure 1b). With r relays, $(r+1)n^2$ messages are sent.

DA2A implementation. DA2A has two modes: normal and alert. Every instance of DA2A starts in the normal mode, in which a broadcast uses direct all-to-all and also informs the master of the broadcast. A *detect*() operation proceeds follows:

- Wait 2Δ time for all players to inform it of their broadcasts.
- In case inform messages are missing from some subset of players $P \subset \Pi$, *detect*() returns P.

- Otherwise, the master waits 2Δ time to make sure that all messages that had been sent have arrived, and then queries all players if they deliver messages from all players.
- If none of the players complains, *detect*() returns $\{\}$.
- Otherwise, the master picks a player p_i that did not deliver a message from player p_j and instructs all players to switch to the alert mode in which they re-broadcast their messages using relayed all-to-all with $2f+1$ players different from p_i and p_j acting as relays.
- After waiting 2Δ time, the master again queries all players if they deliver messages from all players. For every two players p_j and p_i s.t. p_i does not deliver a message from p_j, the master asks the relays whether they received a message from p_j. The relays' replies are signed and used as proof of a deviation. In case $f+1$ relays say yes, the return set includes p_i. Otherwise, it includes p_j.

Correctness. We now prove the detectability property (Definition 1) of our DA2A broadcast.

▶ **Theorem 2.** *If no more than $f+1$ players deviates from the protocol, then (1)* detect() *never returns a player that does not deviate and (2) for every two players p_i, p_j s.t. p_i does not deliver a message from p_j,* detect() *returns either p_i or p_j.*

Proof. Consider two players p_j and p_i s.t. p_i does not deliver a message from p_j in the alert mode. In case $f+1$ relays tell the master that they received a message from p_j, then by the protocol *detect*() includes p_i in its return set, and otherwise it includes p_j. Since p_i does not deliver a message from p_j, we get that either p_i or p_j deviated. Thus, since the master picks $2f+1$ relays other than p_i and p_j, we get that no more than f relays deviate. Therefore, whenever $f+1$ relays report that they received a message from p_j, at least one non-deviating relay forwarded the message from p_j to p_i, meaning that p_i deviated by not delivering it. In addition, since we have $2f+1$ relays, at most f of which deviate, we get at least $f+1$ are not deviating. Therefore, in case fewer than $f+1$ relays report that they received a message from p_j, we get that p_j did not send its message to all relays, i.e., has deviated. ◀

4.2 Sequencing protocol

The sequencing protocol works in *epochs*, where in each epoch every participating player gets an opportunity to append one transaction (or one fixed-size batch of transactions) to the log. To ensure fairness, we commit all the epoch's transactions to the log atomically (all-or-nothing). Recall that we assume that players always have transactions to append.

An *append*(t) operation locally buffers t for inclusion in an ensuing epoch, and waits for it to be sequenced. Each epoch consists of three DA2A communication rounds among players participating in the current epoch (see Figure 2), proceeding as follows:

1. Broadcast a transaction from the local buffer; upon receiving transactions from all, order them by some deterministic rule and sign the hash h of the sequence.
2. Broadcast h; receive from all and verify that all players signed the same hash.
3. Broadcast $\langle commit, epoch, h\rangle$ (signed), and append to local ledger (and return) when receive the same message $f+1$ times.

If any messages are not received, the protocol hangs. The purpose of the first round is to broadcast all the transactions of the epoch. The second round ensures safety; at the end of this round each player validates that all other players signed the same hash of transactions, meaning that only this hash can be committed in the current epoch. The last round ensures recoverability during reconfiguration as we explain in Section 4.3 below. Note that we achieve fairness by waiting for all players; an epoch is committed only if all the players sign the same hash, and since each player signs a hash that contains its own transaction, we get that either all the players' transactions appear in the epoch, or the epoch is not committed.

Read operations. Since all players make progress together, they all have up-to-date local copies of the ledger. A $read(l)$ operation simply returns the last l committed transactions in the local ledger. To make sure byzantine players do not lie about committed transactions, a returned batch of transactions st for epoch k is associated with a *proof*, which is either (1) a newConfig message from the master that includes st (more details below), or (2) $f+1$ epoch k round 3 messages, each of which contains a hash of st.

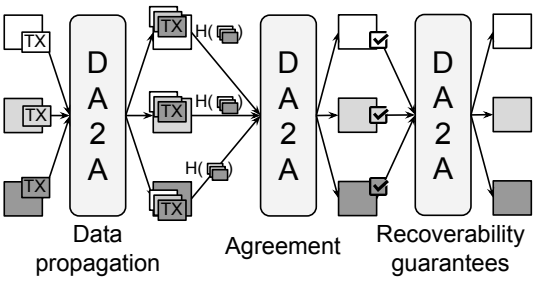


Figure 2 Sequencing protocol.

Asynchronous broadcast. The first round of our sequencing protocol exchanges transactions (data), the second round exchanges hashes of the transactions (meta-data), and the last round exchanges commit messages (meta-data). Hence, the first round consumes most of the bandwidth. In order to increase throughput, we decouple data from meta-data and asynchronously broadcast transactions (i.e., execute the first round) of every epoch as soon as possible. However, in order to be able to validate transactions, we perform rounds 2 and 3 sequentially.

In other words, we divide our communication into a data path and a meta-data path, where the data path is out-of-order and the meta-data path orders the data. This is a common approach, used, for example, in atomic broadcast algorithms that use reliable broadcast to exchange messages and a consensus engine to order them [13, 20].

4.3 Recovery

To detect deviations that prevent progress, we use the $detect()$ operation exposed by DA2A. Recall that the sequencing protocol is an infinite sequence of DA2A instances. Therefore, the master sequentially invokes detect() operations in all DA2A instances. If it returns a non-empty set S, the master invokes reconfiguration.

During reconfiguration the master first stops the current configuration and learns its closing state by sending a reconfig message to the current committee. To prove to the players on the committee that a reconfiguration is indeed necessary, the master attaches to the reconfig message proof reconfiguration is warranted. This can be evidence of active deviation, or a proof of passive deviation returned from DA2A $detect()$. When a player receives a reconfig message, it validates the proof for the reconfiguration, sends its local state (ledger) to the master, and waits for a newConfig message from the master. When a player receives newConfig with a new configuration, it validates that every player removal is justified by a proof, and ignores requests that do not have a valid proof.

State transfer. Note that while a byzantine player cannot make the master believe that an uncommitted epoch has been committed (a committed epoch must be signed by all the epoch's players), it can omit a committed epoch when asked (by the master) about its local

state. Such behavior, if not addressed, could potentially lead to a safety violation: suppose that some byzantine player p does not broadcast its last message in the third round in epoch k, but delivers messages from all other players. In this case, p has proof that epoch k is committed, and may return these transactions in response to a read. However, no other player has proof that epoch k is committed and p withholds epoch k's commit from the master. In this case, the new configuration will commit different transactions in epoch k, which will lead to a safety violation when a *read* operation will be performed.

The third round of the epoch is used to overcome this potential problem. If the master observes that some player receives all messages in the second round of epoch k, it concludes that some byzantine player *may* have committed this epoch. Therefore, in this case, the master includes epoch k in the closing state. Since the private keys of byzantine players are unavailable to the master, it signs the epoch with its own private key, and sends it to all players in the new configuration (committee) as the opening state. A player that sees an epoch with the master's signature refers to it as if it is signed by all players. (Recall that the master is a trusted entity, emulated by a BFT protocol.)

4.4 Rationality – proof sketch

We now informally argue that following the protocol is an equilibrium for all rational committee players. The formal proof of appears in the full paper [35].

Since a round 2 message is required from all committee members in order for an epoch to be committed, and since no committee member will sign a hash on a sequence that excludes its transaction (otherwise its ratio in the ledger will decrease), we get that a player on the committee cannot be excluded from a committed epoch. Therefore, players cannot increase their ratio in the ledger by active deviation. Moreover, since the master may punish them for an active deviation by removing them from the committee, following the protocol dominates any active deviation.

As for passive deviations, a possible strategy for a rational player p_i is to try to "frame" another player p_j and get it removed by the master, in which case p_i's ratio in the ledger will grow. It can try to do this by not sending messages to p_j or by lying about not delivering p_j's messages. In order to prove Nash equilibrium we need to show that if all rational players but a player p_i follow the protocol, then even if all f byzantine players help p_i (and so $f + 1$ players deviate from the protocol), p_i still cannot frame another player and get it removed: This follows from Theorem 2.

Moreover, since we assume that among ledgers with the same ratio players prefer longer ones, sending protocol messages as fast as possible dominates slower sending.

5 FairLedger implementations

We implement FairLedger based on Iroha's framework, written in C++. For better comparison we only change Iroha's consensus algorithm (called Sumeragi [46]) with our sequencing protocol, while keeping other components almost untouched (e.g., cryptographic components, communication layer, and client API). This implementation is described in Section 5.1.

In order to evaluate the FairLedger protocol itself, independently of the Hyperledger framework, we implement another version of FairLedger's sequencing protocol based on PBFT's code structure, written in C++ as well, as described in Section 5.2.

5.1 Hyperledger implementation

The Hyperledger framework consists of two types of entities, *players* (committee members in our case) that run the protocol, and *clients* that generate transactions and send them to players for sequencing.

The FairLedger protocol at each player is orchestrated by a single thread, referred to as *logic thread*. The logic thread receives transactions from clients as well as messages from other players into a wait-free incoming event queue. The connections between clients and players are implemented as GRPC sessions [30] (internally using TCP) sending Protobuf messages [29]. The logic thread maintains a map of epoch numbers to epoch states. An epoch state consists of verified events of that epoch, one event slot per player.

Upon receiving a new message, the logic thread verifies it and decides based on the epoch state whether it needs to broadcast a message to other players. Whenever broadcast is required, the logic thread creates and signs the new message, determines the set of its destinations (based on the epoch state), and creates send-message tasks, one per destination. These tasks are handed over to a work-stealing thread pool, in which each thread communicates with its destination over a GRPC connection (See Figure 3).

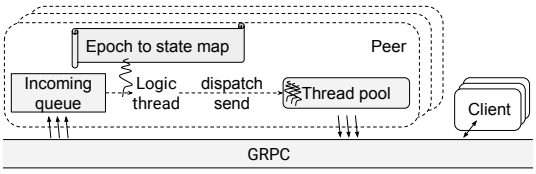


Figure 3 FairLedger implementation in Hyperledger.

Iroha is built in a modular fashion, which allows us to swap Sumeragi with FairLedger in a straightforward way. Our evaluation (in Section 6.2) shows that additional Iroha components beyond the consensus engine adversely affect performance. Yet, these components are essential for Hyperledger. For example, Iroha supports multiple operating systems (including Android and iOS) and can be activated from java script code (via a web interface). Such features are essentials for client-facing systems like Iroha, and using standard libraries such as GRPC enables simple and clean development, which is less prone to bugs.

5.2 Standalone implementation

To eliminate the effect of the overhead induced the Hyperledger framework, we further evaluate the FairLedger protocol by itself, independently of the additional components. To this end, we employ the PBFT code [17] as our baseline. PBFT uses UDP channels, and is almost entirely self-contained, it depends only on one external library, for cryptography.

In this implementation of FairLedger, the logic thread directly communicates with clients and players over UDP. As in our Hyperledger implementation, the logic thread uses a map of epoch numbers to epoch states, and follows the same logic for generating messages.

Using UDP requires us to handle packet loss. We use a dedicated timer thread that wakes up periodically, (after a delay determined according to the line latency), verifies the progress of the minimal unfinished epoch, and requests missing messages from the minimal epoch if needed.

6 Evaluation

We now evaluate our FairLedger protocol using the two prototypes. The Hyperledger prototype is comparable to Iroha, and the standalone prototype is comparable to PBFT.

6.1 Experiment setup

Configuration. We conduct our experiments on Emulab [48]. We allocate 32 servers: 16 Emulab D710 machines for protocol players, and 16 Emulab PC3000 machines for request-generating threads (clients). Each D710 is a standard machine with a 2.4 GHz 64-bit Quad Core Xeon E5530 Nehalem processor, and 12 GB 1066 MHz DDR2 RAM. Each PC3000 is a single 3GHz processor machine with 2GB of RAM.

Given that our system is intended for deployment over WAN among financial institutions, we configure the network latency among players to 20ms. In Emulab, the communication takes place over a shared 1Gb LAN, denoted S-LAN. Each client is connected to a single (local) player with a zero latency 1Gb LAN. In case clients need to communicate directly with remote players (as they do in Iroha's design), they do so over S-LAN, i.e., with a latency penalty. We benchmark the system at its throughput saturation point.

In our Hyperledger prototype evaluation, we use version v0.75. Since in normal mode we assume no byzantine behavior, we configure Iroha with no faulty players, so it signs each transaction once. The request-generating threads create transactions formatted according to Iroha's specification (given in Protobuf), which consists of a few hundreds of bytes of data.

In our standalone prototype evaluation, we create packets of a similar size, namely 512B of data, as this is the transaction size in our expected use case.

Test scenarios. We compare Iroha and PBFT to FairLedger's two operation modes – the failure-free normal mode and the alert mode activated in case of attacks.

We evaluate the alert mode both under attack of a single byzantine player, and without an attack. In the alert mode we assume that f=1, and hence employ 3 relays. In the attack scenario the byzantine player remains undetectable by the master. Specifically, one of the relays withholds messages that it needs to send to one of the other relays.

6.2 Hyperledger

In order to deal with f failures, FairLedger needs $2f+3$ players, and Iroha needs $3f+1$. Therefore, we scale our evaluation from 5 to 9 players. Iroha's clients perform asynchronous operations, and so the operation latency is always zero. Hence, we focus this comparison on throughput.

Figure 4 compares the two modes of FairLedger with Iroha. Results show that FairLedger's normal mode has much higher throughput (up to 3.5x) than Iroha's and the difference grows with the number of players. In both algorithms, due to the usage of GRPC, the bottleneck is the broadcast. FairLedger commits more transactions per broadcast, since each epoch consists of one message from every player, whereas Iroha pays the cost of broadcast for every client request. Therefore, Iroha suffers more as the broadcast cost increases (as we have more players to send messages to).

FairLedger's alert modes incur a 44% reduction in throughput with 5 players, and even more as the number of players increases, because the relays worsen the bottleneck by issuing additional broadcast operations. Byzantine behavior slightly improves performance since withholding messages reduces the load on the relays. However, this effect is negligible.

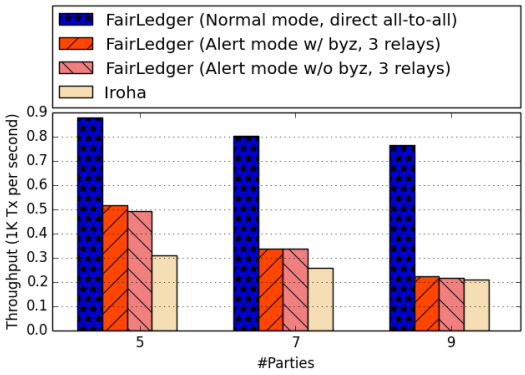


Figure 4 Throughput of FairLedger and Iroha over simulated WAN.

6.3 Standalone prototype

We evaluate our FairLedger prototype that is based on PBFT's code structure. We configure PBFT parameters in a way that maximizes PBFT's throughput, enabling batching and enough outstanding client-requests to saturate the system. We indeed achieve similar results to those reported in recent work running PBFT over WAN [40]. Again, since in order to deal with f failures PBFT requires $3f+1$ players and FairLedger $2f+3$, we run the evaluation with 7 to 16 players. Figure 5 shows the throughput and latency achieved by the protocols.

First, we observe that the absolute throughput is 5x higher than with Iroha. This is thanks to PBFT's optimized bare-metal approach, which sacrifices modularity and maintainability for raw performance. We further see that FairLedger's normal mode has higher throughput than PBFT. This is because PBFT's clients are directed to a single player (referred to as primary or leader), while FairLedger's clients address their nearest player, distributing the load evenly among them.

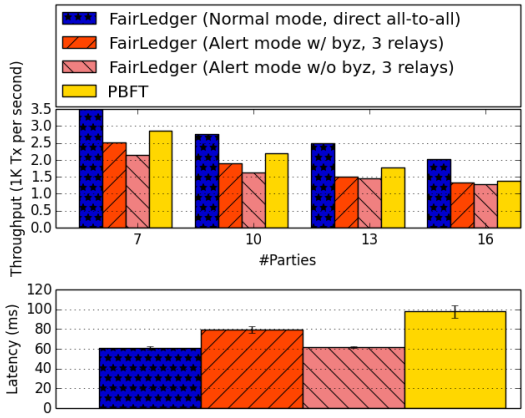


Figure 5 Throughput and latency of FairLedger and PBFT over simulated WAN.

FairLedger's alert mode with three relays reduces throughput by 30%-40% compared to the normal mode. Note that with 7 players, PBFT achieves about 16% higher throughput than FairLedger's alert mode, but as the number of players increases, the gap closes, reaching 9% lower throughput than PBFT's with 16 players.

We measure latency below the saturation point. The results for all configuration sizes are similar, and so we depict in Figure 7 only the results with 10 nodes. Error bars depict the standard deviation. The average latency of FairLedger clients in the normal mode is 64ms, which is close to the network latency of 3 rounds of 20ms. Indeed when communicating over WAN, the performance penalty of signing and verifying signatures is negligible. PBFT's average latency is about 106ms, and consists of 3 PBFT rounds and 2 client-primary communication steps.

The average latency of FairLedger's alert mode with a byzantine relay is 86ms, since it consists of 4 rounds of communication. The reason is that one player is always one round behind the rest due to missing the byzantine player's message. Since in the third round he require messages from $f+1$ players (and not all of them), there is no need to wait for the lagging player's round 3 message, and the epoch ends after 4 rounds. The latency of the alert modes without byzantine players is 64ms, similarly to the normal mode.

7 Related Work

Fairness and rationality. Our work is indebted to recent works that combine game theory and distributed systems [2, 3, 5, 9, 24, 25, 36, 41, 47] to implement different cooperative services. In particular, we adopt a BAR-like model [5, 36, 41]. As in previous works on BAR fault tolerance [5, 36], we assume non-colluding rational players, whereas colluding players are deemed byzantine. As in [41], we do not assume altruistic players – all non-byzantine players are rational in our model.

Practical byzantine fault tolerant consensus protocols [1, 6–8, 15, 16, 18, 23, 32, 37–40, 49] have been studied for more than two decades, but to the best of our knowledge, only three consider some notion of fairness [7, 9, 40], and only one of which deals with rational players [9].

One of the important insights in Prime [7] is that the freedom of the leader to propose transactions must be restricted and verified by other participants. To this end, Prime extends PBFT [16] with three additional all-to-all communication rounds at the beginning, in which participants distribute among them self transactions they wish to append to the ledger. The leader proposes in round 4 a batch of transactions that includes all sets of transactions it gets in round 3 from $2f + 1$ participants. Since each transaction proposed by some participant is passed to the leader by at least $2f + 1$ participants, its participant may expect its transaction to be proposed. In case a participant send a request and the leader does not propose it for some time T, the participant votes to replace the leader. As a result, Prime guarantees that during synchronous periods every transaction is committed in a bounded time T.

Similarly to FairLedger, Prime uses batching to commit transactions of different participants atomically together, and uses a PKI to ensure fairness and provide proofs that the batches are valid. However, their fairness guarantee is weaker than ours. Since the first three rounds are asynchronous (i.e., participants do not wait to hear from all, but rather echo messages as soon as they receive them), there is no bound on the ratio of transactions issued by different participants that are committed during T. More importantly, Prime assumes that all non-byzantine participants follow the protocol, and we do not see a simple way to adjust to overcome rational behavior. For example, there is no incentive for participants to echo transactions issued by other participants in the first three rounds; to the contrary – the less they echo, the less transactions from other participants will be proposed by the leader.

Honeybadger [40] is a recent protocol for permissioned blockchians, which is built on top of an optimization of the atomic broadcast algorithm by Cachin et al. [13]. It works under fully asynchronous assumptions and provides probabilistic guarantees. Honeybadger assumes

a model with n servers and infinitely many clients. In brief, clients submit transactions to all the servers, and servers agree on their order in epochs. In each epoch, participants pick a batch of transactions (previously submitted to them by clients) and use an efficient variation of Bracha's reliable broadcast [11] to disseminate the batches. Then, participants use a randomized binary consensus algorithm by Ben-Or et al. [10] for every batch to agree whether or not to include it in the epoch.

Similarly to FairLedger, they use epochs to batch transactions proposed by different players, and commit them atomically together. Their (probabilistic) fairness guarantee is stronger than the one in Prime: they bound the number of epochs (and accordingly the number of transactions) that can be committed before any transaction that is successfully submitted to $n - f$ servers. However, if we adapt their protocol to our model where we do not consider clients and require fairness among players, we observe that their guarantee is weaker than ours: Since communication is asynchronous, it may take arbitrarily long for a transaction by player p_i to get (be submitted) to $n - f$ players, and in the meantime, other players may commit an unbounded number of transactions. In addition, their protocol uses building blocks (e.g., Bracha's broadcast [11] and Ben-Or et al. [10] randomized consensus) that are not designed to deal with rational behavior. Moreover, rational players that wish to increase their ratio in the ledger will not include transactions issued by other players in their batches.

The only practical work that deals with rational players we are aware of is Helix [9]. However, in contrast to our work, Helix provide only probabilistic fairness guarantees and relies on a randomness beacon.

Finally, it worth noting that Prime, Honeybadger, and Helix are much more complex than FairLedger. Prime's and Helix's description in [7] and [9], respectively, is spread over more than 6 double column pages, and the reader is referred to their full paper versions for more details. Honeybadger combines several building blocks (e.g., the atomic broadcast by Cachin et al. [13]), each of which is complex by itself.

BFT protocols and assumptions. The vast majority of the practical BFT protocols [6,8,23, 32,37–39,49], staring with PBFT [16] assume a model with n symmetric servers (participants) that communicate via reliable eventually synchronous channels. Therefore, they can tolerate at most $f < n/3$ byzantine failures [26], and cannot accurately detect participants' passive deviations (withholding a message or lying about not receiving it); intuitively, it is impossible to distinguish whether a player maliciously withholds its message or the message is just slow. Since passively deviating participants cannot be accurately detected, they cannot be punished or removed, and thus byzantine participants can forever degradate performance [18], and rational behavior cannot be disincentivize.

We, in contrast, assume synchronous communication, which together with the use of a PKI allows FairLedger to be simple, tolerate almost any minority of byzantine failures, guarantee fairness, detect passive as well as active deviations, and penalize deviating players. FairLedger uses the synchrony bound only to detect and remove byzantine players that prevent progress, allowing it to be very long (even minutes) without hurting normal case performance. To reduce the cost of using a PKI, FairLedger signs only the hashes of the messages. Moreover, in WAN networks the cost of PKI is reduced due to longer channels delays.

As illustrated by works on Prime [7] and Aardvark [18] most BFT protocols are vulnerable to performance degradation caused by byzantine participants. To remedy this, Aardvark focuses on improving the worst case scenario. We, on the other hand, follow the approach

taken in Zyzzyva [32], and optimize the failure-free scenario. We take this approach because byzantine failures are rare in financial settings, and one can expect break-ins to be investigated remedied.

We implement FairLedger inside Iroha [45], which is part of the Hyperledger [28] project. Specifically, we substitute the ledger protocol in Iroha, which was originally based on the BFT protocol in BChain [23], with FairLedger. In brief, their protocol consists of a chain of $3f + 1$ participants, where the first $f + 1$ order transactions. To deal with a passively deviating participant that withholds messages in the chain, they transfer both the sender and the receiver (although only one of them deviates from the protocol) to the back of the chain, where they do not take part in ordering transactions. Similarly to FairLedger, they assume synchrony with coarse time bounds and use it to detect passive deviations. However, in contrast to FairLedger, they do no accurately detect byzantine players and punish correct ones as well. Moreover, since the head of the chain decides on the transaction order, Iroha does not guarantee fairness.

Broadcast primitives. In order to detect passive deviation we define DA2A, a new detectable all-to-all communication abstraction. Even though many practical byzantine broadcasts [12–14, 20, 22, 27, 43] were proposed in the past, DA2A is the first to extend its API with a *detect*() method, which accurately returns all misbehaving players.

8 Discussion

Blockchains are widely regarded as the trading technology of the future; industry leaders in finance, banking, manufacturing, technology, and more are dedicating significant efforts towards advancing this technology. The heart of a blockchain is a distributed shared ledger protocol. In this paper, we developed FairLedger, a novel shared ledger protocol for the blockchain setting. Our protocol features the first byzantine fault-tolerant consensus engine to ensure fairness when all players are rational. It is also simple to understand and implement. We integrated our protocol into Hyperledger, a leading industry blockchain for business framework, and showed that it achieves superior performance to existing protocols therein. We further compared FairLedger to PBFT in a WAN setting, achieving better results in failure-free scenarios.

References

1. Michael Abd-El-Malek, Gregory R Ganger, Garth R Goodson, Michael K Reiter, and Jay J Wylie. Fault-scalable Byzantine fault-tolerant services. In *Operating Systems Review*, 2005.
2. Ittai Abraham, Lorenzo Alvisi, and Joseph Y Halpern. Distributed computing meets game theory: combining insights from two fields. *Acm Sigact News*, 2011.
3. Ittai Abraham, Danny Dolev, and Joseph Y Halpern. Distributed protocols for leader election: A game-theoretic perspective. In *International Symposium on Distributed Computing*, 2013.
4. Ittai Abraham and Dahlia Malkhi. BVP: Byzantine Vertical Paxos, 2016.
5. Amitanand S Aiyer, Lorenzo Alvisi, Allen Clement, Mike Dahlin, Jean-Philippe Martin, and Carl Porth. BAR fault tolerance for cooperative services. In *operating systems review*, 2005.
6. Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. Customizable fault tolerance forwide-area replication. In *Reliable Distributed Systems, 2007. SRDS 2007.*, 2007.
7. Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. Prime: Byzantine replication under attack. *IEEE Transactions on Dependable and Secure Computing*, 2011.

8 Yair Amir, Claudiu Danilov, Jonathan Kirsch, John Lane, Danny Dolev, Cristina Nita-Rotaru, Josh Olsen, and David Zage. Scaling byzantine fault-tolerant replication towide area networks. In *Dependable Systems and Networks, 2006. DSN 2006*, 2006.

9 Avi Asayag, Gad Cohen, Ido Grayevsky, Maya Leshkowitz, Ori Rottenstreich, Ronen Tamari, and David Yakira. A Fair Consensus Protocol for Transaction Ordering. In *A Fair Consensus Protocol for Transaction Ordering*, pages 55–65, 2018. doi:10.1109/ICNP.2018.00016.

10 Michael Ben-Or, Boaz Kelmer, and Tal Rabin. Asynchronous secure computations with optimal resilience. In *PODC*. ACM, 1994.

11 Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 1987.

12 Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, 1985.

13 Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Cryptology*. Springer, 2001.

14 Christian Cachin and Stefano Tessaro. Asynchronous verifiable information dispersal. In *Reliable Distributed Systems, 2005. SRDS 2005. 24th IEEE Symposium on*. IEEE, 2005.

15 Ran Canetti and Tal Rabin. Fast asynchronous Byzantine agreement with optimal resilience. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, 1993.

16 Miguel Castro, Barbara Liskov, et al. Practical Byzantine fault tolerance. In *OSDI*, 1999.

17 Miguel Castro, Barbara Liskov, et al. BFT - Practical Byzantine Fault Tolerance (software). http://www.pmg.csail.mit.edu/bft/#sw, 2017. [Online; accessed 16-Apr-2017].

18 Allen Clement, Edmund L Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In *NSDI*, 2009.

19 CoreOS. etcd – a highly-available key value store for shared configuration and service discovery. https://coreos.com/etcd/, 2017.

20 Flaviu Cristian, Houtan Aghili, Raymond Strong, and Danny Dolev. *Atomic broadcast: From simple message diffusion to Byzantine agreement*. Citeseer, 1986.

21 Danny Dolev and H. Raymond Strong. Authenticated algorithms for Byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

22 Vadim Drabkin, Roy Friedman, and Alon Kama. Practical Byzantine group communication. In *ICDCS*. IEEE, 2006.

23 Sisi Duan, Hein Meling, Sean Peisert, and Haibin Zhang. BChain: Byzantine Replication with High Throughput and Embedded Reconfiguration. In *OPODIS 2014*. OPODIS 2014, 2014.

24 Joan Feigenbaum, Christos Papadimitriou, and Scott Shenker. Sharing the cost of muliticast transmissions (preliminary version). In *Theory of computing*. ACM, 2000.

25 Michal Feldman, Christos Papadimitriou, John Chuang, and Ion Stoica. Free-riding and whitewashing in peer-to-peer systems. *Journal on Selected Areas in Communications*, 2006.

26 Michael J Fischer, Nancy A Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1(1):26–39, 1986.

27 Matthias Fitzi and Martin Hirt. Optimally efficient multi-valued byzantine agreement. In *PODC*. ACM, 2006.

28 The Linux Foundation. Hyperledger. https://www.hyperledger.org/.

29 Google. Protocol Buffers - data interchange format. https://github.com/google/protobuf.

30 Google. GRPC – Open-source universal RPC framework. http://www.grpc.io/, 2017.

31 Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX technical conference*, 2010.

32 Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *ACM SIGOPS Operating Systems Review*, 2007.

33 Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In *PODC*. ACM, 2009.

34 Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

35 Kfir Lev-Ari, Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi. FairLedger: A Fair Blockchain Protocol for Financial Institutions. *arXiv preprint arXiv:1906.03819*, 2019.

36 Harry C Li, Allen Clement, Edmund L Wong, Jeff Napper, Indrajit Roy, Lorenzo Alvisi, and Michael Dahlin. BAR gossip. In *Operating systems design and implementation*, 2006.

37 Jinyuan Li and David Maziéres. Beyond One-Third Faulty Replicas in Byzantine Fault Tolerant Systems. In *NSDI*, 2007.

38 Shengyun Liu, Christian Cachin, Vivien Quéma, and Marko Vukolic. XFT: practical fault tolerance beyond crashes. *CoRR, abs/1502.05831*, 2015.

39 J-P Martin and Lorenzo Alvisi. Fast byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, 2006.

40 Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of BFT protocols. In *CCS*. ACM, 2016.

41 Thomas Moscibroda, Stefan Schmid, and Rogert Wattenhofer. When Selfish Meets Evil: Byzantine Players in a Virus Inoculation Game. In *PODC*. ACM, 2006.

42 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

43 Michael Reiter. The Rampart toolkit for building high-integrity services. *Theory and Practice in Distributed Systems*, 1995.

44 Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 1978.

45 Soramitsu. Iroha - A simple, decentralized ledger. `http://iroha.tech/en/`.

46 Soramitsu. Sumeragi - a Byzantine Fault Tolerant consensus algorithm. `https://github.com/hyperledger/iroha/blob/master/docs/iroha_whitepaper.md`, 2017.

47 Vikram Srinivasan, Pavan Nuggehalli, Carla-Fabiana Chiasserini, and Ramesh R Rao. Cooperation in wireless ad hoc networks. In *INFOCOM*. IEEE, 2003.

48 Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *OSDI*, 2002.

49 Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from execution for byzantine fault tolerant services. *OSR*, 2003.

Deconstructing Stellar Consensus

Álvaro García-Pérez
IMDEA Software Institute, Madrid, Spain

Maria A. Schett
University College London, United Kingdom

Abstract

Some of the recent blockchain proposals, such as Stellar and Ripple, allow for open membership while using quorum-like structures typical for classical Byzantine consensus with closed 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 consensus protocols underlying such blockchains are poorly understood, and their correctness has not been rigorously investigated. In this paper we rigorously prove correct the Stellar Consensus Protocol (SCP), with our proof giving insights into the protocol structure and its use of lower-level abstractions. To this end, we first propose an abstract version of SCP that uses as a black box Stellar's federated voting primitive (analogous to reliable Byzantine broadcast), previously investigated by García-Pérez and Gotsman [7]. The abstract consensus protocol highlights a modular structure in Stellar and can be proved correct by reusing the previous results on federated voting. However, it is unsuited for realistic implementations, since its processes maintain infinite state. We thus establish a refinement between the abstract protocol and the concrete SCP that uses only finite state, thereby carrying over the result about the correctness of former to the latter. Our results help establish the theoretical foundations of decentralised blockchains like Stellar and gain confidence in their correctness.

2012 ACM Subject Classification Theory of computation → Distributed computing models

Keywords and phrases Blockchain, Consensus protocol, Stellar, Byzantine quorum systems

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.5

Related Version `http://arxiv.org/abs/1911.05145`

Funding This work was supported by ERC Starting Grant RACCOON, Spanish State Research Agency project BOSCO, and Madrid Regional Government project BLOQUES.

Acknowledgements We thank Ilya Sergey, Giuliano Losa and Alexey Gotsman for their comments.

1 Introduction

Permissioned blockchains are becoming increasingly popular due to the low-energy consumption and hard guarantees they provide on when a transaction can be considered successfully committed. Such blockchains are often based on classical Byzantine fault-tolerant (BFT) consensus protocols, like PBFT [4]. 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) or by a more general structure of a Byzantine quorum system (BQS) [12]. Unfortunately, defining quorums in this way requires fixing the number of participants in the system, which prevents decentralisation.

Some of the recent blockchain proposals, such as Stellar [13] and Ripple [15], allow for open membership while using quorum-like structures typical for classical Byzantine consensus with closed membership. This is achieved by constructing quorums 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)*, where each participant selects a set of *quorum*

© Álvaro García-Pérez and Maria A. Schett;
licensed under Creative Commons License CC-BY
23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller; Article No. 5; pp. 5:1–5:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

slices – sets of nodes each of which would convince the participant to accept the validity of a given statement (§2). 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 [1, 2]. Unfortunately, even though Stellar has been deployed as a functioning blockchain, the structure of the consensus protocol underlying it is poorly understood and its correctness has not been rigorously investigated. In this paper we aim to close this gap, rigorously defining and proving correct the Stellar Consensus Protocol (SCP). Apart from giving more confidence in the correctness of the protocol, our proof is structured in such a way as to give insights into its structure and its use of lower-level abstractions.

In more detail, the guarantees provided by SCP are nontrivial. When different participants in an FBQS choose different slices, only a subset of the participants may take part in a subsystem in which every two quorums intersect in a correct node – a property required for achieving consensus. The system may partition into such subsystems, and SCP will guarantee agreement within each of them. In blockchain terms, the blockchain may fork, but in this case each fork will be internally consistent, a property that is enough for business applications of the Stellar blockchain. The subsystems where agreement is guaranteed are characterised by Mazières et al. [14] through the notion of *intact sets*. Our proof of correctness establishes safety and liveness properties of SCP relative to such intact sets (§3).

As a stepping stone in the proof, we first propose an *abstract version* of SCP that uses as a black box Stellar's federated voting primitive (analogous to reliable Byzantine broadcast) previously investigated by García-Pérez and Gotsman [7] (§5). This abstract formulation allows specifying the protocol concisely and highlights the modular structure present in it. This allows proving the protocol by reusing the previous results on federated voting [7] (reviewed in §4). However, the abstract protocol is unsuited for realistic implementations, since its processes maintain infinite state. To address this, we formulate a realistic version of the protocol – a *concrete SCP* – that uses only finite state. We then prove a refinement between the abstract and concrete SCP, thereby carrying over the result about the correctness of former to the latter (§6).

A subtlety in SCP is that its participants receive information about quorum slices of other participants directly from them. Hence, Byzantine participants may lie to others about their choices of quorum slices, which may cause different participants to disagree on what constitutes a quorum. Our results also cover this realistic case (§7).

Overall, our results help establish the theoretical foundations of decentralised blockchains like Stellar and gain confidence in their correctness. Due to space constraints, proofs are deferred to an extended version of the paper [8].

2 Background: System Model and Federated Byzantine Quorum Systems

System model. We consider a system consisting of a finite *universe* of *nodes* $\mathbf{V}$ and assume a Byzantine failure model where *faulty* nodes can deviate arbitrarily from their specification. All other nodes are called *correct*. Nodes that are correct, or that only deviate from their specification by stopping execution, are called *honest*. Nodes that deviate from their specification in ways other than stopping are called *malicious*. We assume that any two nodes can communicate over an authenticated perfect link. We assume a partial synchronous network, which guarantees that messages arrive within bounded time after some unknown,

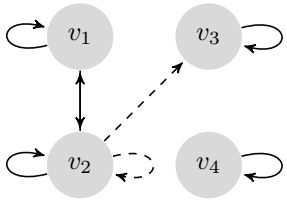
finite *global stabilisation time* (*GST*). Each node has a local timer and a timeout service that can be initialised with an arbitrary delay Δ. We assume that after GST the clock skew of correct nodes is bounded, *i.e.*, after GST two correct nodes can only disagree in the duration of a given delay Δ by a bounded margin.

Federated Byzantine quorum systems. Given a finite universe $\mathbf{V}$ of nodes, a *federated Byzantine quorum system* (*FBQS*) [13, 7] is a function $\mathcal{S} : \mathbf{V} \to 2^{2^{\mathbf{V}}} \setminus \{\emptyset\}$ that specifies a non-empty set of *quorum slices* for each node, ranged over by q. We require that a node belongs 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 node. A non-empty set of nodes $U \subseteq \mathbf{V}$ is a *quorum* in an FBQS $\mathcal{S}$ iff U contains a slice for each member, *i.e.*, $\forall v \in U. \exists q \in \mathcal{S}(v). q \subseteq U$.

For simplicity, for now we assume that faulty nodes do not equivocate about their quorum slices, so that all the nodes share the same FBQS. In §7 we consider the more realistic *subjective FBQS* [7], where malicious nodes may lie about their slices and different nodes have different views on the FBQS. There we also lift the results on the subsequent sections of the paper to subjective FBQSes.

▶ **Example 1.** Consider a universe $\mathbf{V}$ with $3f + 1$ nodes, and consider the FBQS $\mathcal{S}$ where for every node $v \in \mathbf{V}$, the set of slices $\mathcal{S}(v)$ consists of every set of $2f + 1$ nodes that contains v itself. $\mathcal{S}$ encodes the classical cardinality-based quorum system of $3f + 1$ nodes with failure threshold f, since every set of $2f + 1$ or more nodes is a quorum.

▶ **Example 2.** Let the universe $\mathbf{V}$ contain four nodes v_1 to v_4, and consider the FBQS $\mathcal{S}$ in the diagram below.



$$\begin{aligned}
\mathcal{S}(v_1) &= \{\{v_1, v_2\}\} \\
\mathcal{S}(v_2) &= \{\{v_1, v_2\}, \{v_2, v_3\}\} \\
\mathcal{S}(v_3) &= \{\{v_3\}\} \\
\mathcal{S}(v_4) &= \{\{v_4\}\}
\end{aligned}$$

For each node, all the outgoing arrows with the same style determine one slice. Node v_2 has two slices, determined by the solid and dashed arrow styles respectively. The rest of the nodes have one slice. $\mathcal{S}$ has the following set of quorums $\mathcal{Q} =$

$$\{\{v_1, v_2\}, \{v_2, v_3\}, \{v_3\}, \{v_4\}, \{v_1, v_2, v_3\}, \{v_3, v_4\}, \{v_1, v_2, v_4\}, \{v_2, v_3, v_4\}, \{v_1, v_2, v_3, v_4\}\}.$$

A consensus protocol that runs on top of an FBQS may not guarantee global agreement, because when nodes choose slices independently, only a subset of the nodes may take part in a subsystem in which every two quorums intersect in at least one correct node – a basic requirement of a Byzantine quorum system [12] to ensure agreement. To formalise which parts of the system may reach agreement internally, we borrow the notions of *intertwined nodes* and of *intact set* from [14]. Two nodes v_1 and v_2 are *intertwined* iff they are correct and every quorum containing v_1 intersects every quorum containing v_2 in at least one correct node. Consider an FBQS $\mathcal{S}$ and a set of nodes I. The *projection* $\mathcal{S}|_I$ of $\mathcal{S}$ to I is the FBQS over universe I given by $\mathcal{S}|_I(v) = \{q \cap I \mid q \in \mathcal{S}(v)\}$. For a given set of faulty nodes, a set I is an *intact set* iff I is a quorum in $\mathcal{S}$ and every member of I is intertwined with each other in the projected FBQS $\mathcal{S}|_I$. The intact sets characterise those sets of nodes that can reach consensus, which we later show using the following auxiliary result.

▶ **Lemma 3.** *Let $\mathcal{S}$ be an FBQS and assume some set of faulty nodes. Let I be an intact set in $\mathcal{S}$ and consider any two quorums U_1 and U_2 in $\mathcal{S}$ such that $U_1 \cap I \neq \emptyset$ and $U_2 \cap I \neq \emptyset$. Then the intersection $U_1 \cap U_2$ contains some node in I.*

The maximal intact sets are disjoint with each other:

▶ **Lemma 4.** *Let $\mathcal{S}$ be an FBQS and assume some set of faulty nodes. Let I_1 and I_2 be two intact sets in $\mathcal{S}$. If $I_1 \cap I_2 \neq \emptyset$ then $I_1 \cup I_2$ is an intact set in $\mathcal{S}$.*

In SCP the system may split into different partitions – *i.e.*, the maximal intact sets – that may be inconsistent with each other, but which constitute independent systems each of which can reach consensus.

Consider the $\mathcal{S}$ from Example 1, which encodes the cardinality-based quorum system of $3f + 1$ nodes, and let $f = 1$, so that the universe $\mathbf{V}$ contains four nodes v_1 to v_4. If we assume that node v_3 is faulty, then the set $I = \{v_1, v_2, v_4\}$ is the only maximal intact set: I is a quorum in $\mathcal{S}$, and $\mathcal{S}|_I$ contains the quorums $\{\{v_1, v_2\}, \{v_2, v_4\}, \{v_1, v_4\}, \{v_1, v_2, v_4\}\}$, which enjoy quorum intersection. This ensures that every two nodes in I are intertwined in the projected system $\mathcal{S}|_I$.

Now consider the $\mathcal{S}$ from Example 2. If we assume that node v_3 is faulty, then the sets $I = \{v_1, v_2\}$ and $I' = \{v_4\}$ are the maximal intact sets: I and I' are quorums in $\mathcal{S}$, and the projected systems $\mathcal{S}|_I$ and $\mathcal{S}|_{I'}$ enjoy quorum intersection – $\mathcal{S}_I$ contains quorums $\{v_1, v_2\}$ and $\{v_2\}$, and $\mathcal{S}|_{I'}$ contains quorum $\{v_4\}$ – which ensures that every two nodes in either I or I' are intertwined in the projected systems $\mathcal{S}|_I$ and $\mathcal{S}_{I'}$ respectively. It is easy to check that adding any set of correct nodes to either I or I' results in sets that are not quorums in $\mathcal{S}$, or in projected systems that contain some pairs of nodes that are not intertwined.

3 Specifications

Assume a set $\mathbf{Val}$ of *consensus values*. In the consensus protocols that we study in §5–6, each correct node proposes some $x \in \mathbf{Val}$ through an invocation $\mathsf{propose}(x)$, and each node may decide some $x' \in \mathbf{Val}$ through an indication $\mathsf{decide}(x')$. We consider a variant of the *weak Byzantine consensus* specification in [2] that we call *non-blocking Byzantine consensus for intact sets*, which is defined as follows. Given a maximal intact set I,

(*Integrity*) no correct node decides twice,

(*Agreement for intact sets*) no two nodes in I decide differently,

(*Weak validity for intact sets*) if all nodes are honest and every node proposes x, then no node in I decides a consensus value different from x; furthermore, if all nodes are honest and some node in I decides x, then x was proposed by some node, and

(*Non-blocking for intact sets*) if a node v in I has not yet decided in some run of the protocol, then for every continuation of that run in which all the malicious nodes stop, node v eventually decides some consensus value.

The usual *Weak validity* property of consensus [2] ensures that if all nodes are correct and they propose the same consensus value, then no node can decide a consensus value different from the proposed one; and that if all nodes are correct, then a node can only decide a consensus value proposed by some node. Our *Weak validity for intact sets* above adapts this requirement to the nodes in a maximal intact set, and weakens its condition by assuming that all nodes are honest instead of correct. Notice that if every two quorums intersect our property entails the usual one because a correct node is also honest, and because if all nodes are correct then the maximal intact set is the universe. For instance, this condition holds in the cardinality-based quorum systems ($3f + 1$).

The consensus protocols we consider in this paper specify the behaviour of SCP's ballot protocol [13, 14] with one of its suggested strategies for managing timeouts (Strategy 1 from [14]). As discussed in [14], in SCP malicious nodes with good network timing could permanently delay the termination of the nodes in an intact set, and thus the protocol does not provide the usual *Termination* guarantee that every correct node eventually decides some consensus value [2]. Instead, we consider the weaker liveness guarantee of *Non-blocking for intact sets*, which we have obtained by adapting the *Non-blocking* property in [16]. *Non-blocking* requires that some continuation of a given run *exists* in which every correct node terminates. Our *Non-blocking for intact sets* adapts this requirement to the nodes in a maximal intact set, and requires that they terminate in *every* continuation of the run in which malicious nodes are stopped. It is easy to check that if every correct node is in an intact set, then *Non-blocking for intact sets* entails *Non-blocking* in [16]. For instance, this condition holds in the cardinality-based quorum systems $(3f + 1)$. Besides, if every correct node is honest, then *Non-blocking for intact sets* entails the usual *Termination* property that guarantees that every correct node eventually decides some consensus value.

The *non-blocking Byzantine consensus for intact sets* above entails the *weak Byzantine consensus* specification [2] in the cardinality-based quorum systems $(3f+1)$, which guarantees the *Integrity* property above, as well as the usual *Agreement* property that ensures that no two correct nodes decide differently, and the usual *Weak validity* and *Termination* properties that we have recalled in the paragraphs above.

One of the core components of the consensus protocol in §5 is *federating voting* (*FV*) [13, 14]. Assume a set of *voting values* $\mathbf{A}$ that could be disjoint with the set $\mathbf{Val}$ of consensus values (we typically let $\mathbf{A}$ be the set of Booleans $\mathsf{Bool} \equiv \{\mathit{true}, \mathit{false}\}$). FV allows each correct node to vote for some $a \in \mathbf{A}$ through an invocation $\mathsf{vote}(a)$, and each node may deliver some $a' \in \mathbf{A}$ through an indication $\mathsf{deliver}(a')$. The interface of FV is akin to that of consensus, where each node activates itself through the primitive $\mathsf{vote}(a)$. However, FV has weaker liveness guarantees than consensus, which are reminiscent to those of *Byzantine reliable broadcast* from [2] and *weakly reliable Byzantine broadcast* from [7]. Here, we consider a variant of the latter specification that we call *reliable Byzantine voting for intact sets*, which is defined as follows. Given a maximal intact set I,

(*No duplication*) every correct node delivers at most one voting value,

(*Totality for intact sets*) if a node in I delivers a voting value, then every node in I eventually delivers a voting value,

(*Consistency for intertwined nodes*) if two intertwined nodes v and v' deliver a and a' respectively, then $a = a'$, and

(*Validity for intact sets*) if all nodes in I vote for a, then all nodes in I eventually deliver a.

The ability of each node to activate itself independently in the specification above simulates a malicious sender that may send different voting values to each node in the specification of *weakly reliable Byzantine broadcast* from [7].

4 Federated Voting

In this section we recall *federated voting* (*FV*) from [13], which also corresponds to the *Stellar broadcast* considered in [7]. We prove that FV implements the specification of reliable Byzantine voting for intact sets, thereby generalising the results of [7] to the case of multiple intact sets within the system. The consensus protocol that we study in the next section uses multiple instances of FV independent from each other. This is done by letting each node run a distinct process for each instance of FV, which is identified by a *tag* t from some designated set $\mathbf{Tag}$ of tags.

Algorithm 1 below depicts FV over an FBQS $\mathcal{S}$ with set of quorums $\mathcal{Q}$. A node v runs a process federated-voting(v, t) for each tag t. The messages exchanged by such a process are also tagged with t, in order to distinguish them from the messages exchanged for instances of FV associated with tags different from t.

Algorithm 1 Federated voting (FV) over an FBQS $\mathcal{S}$ with set of quorums $\mathcal{Q}$.

```
1  process federated-voting(v ∈ V, t ∈ Tag)
2      voted, ready, delivered ← false ∈ Bool;
3      vote(a)
4          if not voted then
5              voted ← true;
6              send VOTE(t, a) to every v′ ∈ V;
7      when received VOTE(t, a) from every u ∈ U for some U ∈ Q such that
        v ∈ U and not ready
8          ready ← true;
9          send READY(t, a) to every v′ ∈ V;
10     when received READY(t, a) from every u ∈ B for some v-blocking B and
        not ready
11         ready ← true;
12         send READY(t, a) to every v′ ∈ V;
13     when received READY(t, a) from every u ∈ U for some U ∈ Q such that
        v ∈ U and not delivered
14         delivered ← true;
15         trigger deliver(a);
```

FV adapts *Bracha's protocol* for reliable Byzantine broadcast [1], which works over the cardinality-based quorum systems of $3f + 1$ nodes, to the federated setting of the FBQSs. In FV nodes process each other's messages in several stages, where for each tag t progress is denoted by several Boolean flags (line 2 of Algorithm 1). When a node v votes a for tag t for the first time, the node sends VOTE(t, a) to every node (including itself, for uniformity; lines 3–6). When a node v receives a VOTE(t, a) message from a quorum to which v itself belongs, it sends a READY(t, a) message to every node, signalling its willingness to deliver the value a for tag t (lines 7–9). Note that, for each tag t, two nodes in the same intact set I cannot send READY messages with two different voting values through the rule in lines 7–9. Indeed, this would require two quorums of VOTE messages, each with a node in I, with different voting values for the same tag. But by Lemma 3 these quorums would intersect in a node in I, which is by definition correct and cannot send contradictory VOTE messages for the same tag. When a node v receives the message READY(t, a) from a quorum to which v itself belongs, it delivers a for tag t (lines 13–15).

The exchange of READY messages in the protocol is necessary to establish liveness guarantees. It ensures that, if a node in an intact set I delivers a voting value for some tag, other nodes in I have enough information to also deliver a voting value for the same tag. This relies on the rule in lines 10–12, which uses the notion of *v-blocking set* [13]. Given a node v, a set B is *v-blocking* iff B overlaps each of v's slices, *i.e.*, $\forall q \in \mathcal{S}(v).\ q \cap B \neq \emptyset$. (To illustrate this notion, in Example 1 every set of $f + 1$ nodes is v-blocking for every v, and in

Node v_1	Node v_2	Node v_3	Node v_4
vote($false$)	vote($false$)		vote($true$)
VOTE(t, $false$)	VOTE(t, $false$)	VOTE(t, $false$)	VOTE(t, $true$)
READY(t, $false$)	READY(t, $false$)		
			READY(t, $false$)
deliver($false$)	deliver($false$)		deliver($false$)

Figure 1 Execution of the instance of FV for tag t.

Example 2 the set $\{v_1, v_3\}$ is v_2-blocking and the set $\{v_2\}$ is v_1-blocking.) Lines 10–12 allow a node to send a READY(t, a) message even if it previously voted for a different voting value for tag t: this is done if v receives READY(t, a) from each member of a v-blocking set. If v is in an intact set I, the following lemma guarantees that in this case v has received at least one READY(t, a) message from some node in I.

▶ **Lemma 5.** *Let $\mathcal{S}$ be an FBQS and assume a set of faulty nodes. Let I be an intact set in $\mathcal{S}$ and $v \in I$. Then, no v-blocking set B exists such that $B \cap I = \emptyset$.*

By Lemma 5, the first node in I to ever send a READY(t, a) message for a tag t has to do it through the rule in lines 7–9, and hence the value a has been cross-checked by a quorum.

If the condition $v \in U$ in lines 7 and 13 of Algorithm 1 was dropped, this could violate *Agreement for intact sets* as follows. Take the $\mathcal{S}$ from Example 2 and consider a run of FV for some tag t where v_3 is malicious. Node v_3 could respectively send READY(t, a) and READY(t, a') with $a \neq a'$ to correct nodes v_1 and v_2. Since $\{v_3\} \in \mathcal{Q}$, these nodes will respectively deliver a and a' by lines 13–15 of Algorithm 1 without condition $v \in U$.

Our first contribution is to generalise the results of [7] to establish the correctness of FV within each of the maximal intact sets of an FBQS, as captured by Theorem 6 below.

▶ **Theorem 6.** *Let $\mathcal{S}$ be an FBQS and t be a tag. The instance for t of FV over $\mathcal{S}$ satisfies the specification of reliable Byzantine voting for intact sets.*

FV also guarantees the property stated by the following lemma, which helps establish the liveness properties of the consensus protocol that we introduce in §5.

▶ **Lemma 7.** *Let $\mathcal{S}$ be an FBQS and t be a tag. Consider an execution of the instance for t of FV over $\mathcal{S}$. Let I be an intact set in $\mathcal{S}$ and assume that GST has expired. If a node $v \in I$ delivers a voting value then every node in I will deliver a voting value within bounded time.*

We write δ_I for the time that a node in I takes to deliver some voting value after GST and provided that some other node in I already delivered some voting value. The delay δ_I – which is determined by $\mathcal{S}$ and I – is unknown, but Lemma 7 guarantees that it is finite.

▶ **Example 8.** Consider the $\mathcal{S}$ from Example 1, which encodes the cardinality-based quorum system $3f + 1$, and let $f = 1$ such that the universe $\mathbf{V}$ contains four nodes v_1 to v_4. Every set of three or more nodes is a quorum, and every set of two or more nodes is v-blocking for every $v \in \mathbf{V}$. Let us fix a tag t and consider an execution of the instance of FV for tag t where we let the voting values be the Booleans. Assume that nodes v_1, v_2 and v_4 are correct, which constitute the maximal intact set. In the execution, nodes v_1 and v_2 vote *false*, and node v_4 votes *true*. Malicious node v_3 sends the message VOTE$(t, false)$ to every node (highlighted in red) thus helping the correct nodes to deliver *false*.

Figure 1 depicts a possible execution of FV described above, from which a trace can be constructed as follows: all the events in each row may happen concurrently, and any two events in different rows happen in real time, where time increases downwards; in those cells that are

tagged with a message, the node sends the message to every node, and in a given cell a node has received all the messages from every node in the rows above it. (These conventions are only for presentational purposes, and should not be mistaken with the *perfectly synchronised round-based model* of [5], which we do not use.) The quorum $\{v_1, v_2, v_3\}$ sends $\texttt{VOTE}(t, \mathit{false})$ to every node, which makes nodes v_1 and v_2 send $\texttt{READY}(t, \mathit{false})$ to every node through lines 7–9 of Algorithm 1. However, there exists not a quorum U such that $v_4 \in U$ and every member of U sends a message $\texttt{VOTE}(t, a)$ with the same Boolean a, and thus node v_4 sends $\texttt{READY}(t, \mathit{false})$ through lines 10–12 of Algorithm 1, only after receiving corresponding ready messages from the v_4-blocking set $\{v_1, v_2\}$. Observe how node v_4 changes its original vote *true* and sends *false* in the READY message. After every correct node receives $\texttt{READY}(t, \mathit{false})$ from the quorum $\{v_1, v_2, v_4\}$, they all deliver *false*.

5 Abstract Stellar Consensus Protocol

In this section we introduce the *abstract SCP* (*ASCP*), which concisely specifies the mechanism of SCP [13, 14] and highlights the modular structure present in it[1]. Like *Paxos* [9], ASCP uses *ballots* – pairs $\langle n, x\rangle$, where $n \in \mathbb{N}^+$ a natural positive *round number* and $x \in \mathbf{Val}$ a *consensus value*. We assume that **Val** is totally ordered, and we consider a special *null ballot* $\langle 0, \bot\rangle$ and let $\mathbf{Ballot} = (\mathbb{N}^+ \times \mathbf{Val}) \cup \{\langle 0, \bot\rangle\}$ be the set of ballots. (We write $b.n$ and $b.x$ respectively for the round and consensus value of ballot b.) The set **Ballot** is totally ordered, where we let $b < b'$ iff either $b.n < b'.n$, or $b.n = b'.n$ and $b.x < b'.x$.

To better convey SCP's mechanism, we let the abstract protocol use FV as a black box where nodes may hold a binary vote on each of the ballots: we let the set of voting values **V** be the set of Booleans and the set of tags **Tag** be the set of ballots, and let the protocol consider a separate instance of FV for each ballot. A node voting for a Boolean a for a ballot b that carries the consensus value $b.x$ encodes the aim to either *abort* the ballot (when $a = \mathit{false}$) or to *commit* it (when $a = \mathit{true}$) thus deciding the consensus value $b.x$. From now on we will unambiguously use "Booleans", "ballots" and "values" instead of "voting values", "tags" and "consensus values", respectively.

We have dubbed ASCP "abstract" because, although it specifies the protocol concisely, it is unsuited for realistic implementations. On the one hand, each node v maintains infinite state, because it stores a process $\mathsf{federated\text{-}voting}(v, b)$ for each of the infinitely many ballots b in the array $\mathsf{ballots}$ (line 2 of Algorithm 2). On the other hand, each node v may need to send or receive an infinite number of messages in order to progress (lines 6, 8, 15 and 21 of Algorithm 2, which are explained in the detailed description of ASCP below). This is done by assuming a *batched network semantics* (*BNS*) in which the network exchanges *batches*, which are (possibly infinite) sequences of messages, instead of exchanging individual messages: the sequence of messages to be sent by a node when processing an event is batched per recipient, and each batch is sent at once after the atomic processing of the event; once a batch is received, the recipient node atomically processes all the messages in the batch in sequential order. By convention, we let the statement **forall** in lines 7 and 21 of Algorithm 2 consider the ballots b' in ascending ballot order. In §6 we introduce a "concrete" version of SCP that is amenable to implementation, since nodes in it maintain finite state and exchange a finite number of messages; however, this version does not use FV as a black box.

[1] More precisely, in this paper we focus on Stellar's core *balloting* protocol, which aims to achieve consensus. We abstract from Stellar's *nomination* protocol – which tries to converge (best-effort) on a value to propose – by assuming arbitrary proposals to consensus.

Algorithm 2 Abstract SCP (ASCP) over an FBQS $\mathcal{S}$ with set of quorums $\mathcal{Q}$.

```
1  process abstract-consensus(v ∈ V)
2      ballots ← [new process federated-voting(v, b)]^{b∈Ballot};
3      candidate, prepared ← ⟨0, ⊥⟩ ∈ Ballot;
4      round ← 0 ∈ ℕ⁺ ∪ {0};
5      propose(x)
6          candidate ← ⟨1, x⟩;
7          for all b′ ≲ candidate do ballots[b′].vote(false);
8      when triggered ballots[b′].deliver(false) for every b′ ≲ b and prepared < b
9          prepared ← b;
10         if candidate ≤ prepared then
11             candidate ← prepared;
12             ballots[candidate].vote(true);
13     when triggered ballots[b].deliver(true)
14         trigger decide(b.x);
15     when exists U ∈ Q such that v ∈ U and for each u ∈ U exist
        M_u ∈ {VOTE, READY} and b_u ∈ Ballot such that round < b_u.n and either
        received M_u(b_u, true) from u or received M_u(b′, false) from u for every
        b′ ∈ [z_u, b_u) with z_u < b_u
16         round ← min{b_u.n | u ∈ U};
17         start-timer(F(round));
18     when triggered timeout
19         if prepared = ⟨0, ⊥⟩ then candidate ← ⟨round + 1, candidate.x⟩;
20         else candidate ← ⟨round + 1, prepared.x⟩;
21         for all b′ ≲ candidate do ballots[b′].vote(false);
```

ASCP uses the following *below-and-incompatible-than* relation on ballots. We say ballots b and b' are *compatible* (written $b \sim b'$) iff $b.x = b'.x$, and *incompatible* (written $b \not\sim b'$) otherwise, where we let $\bot \neq x$ for any $x \in \mathbf{Val}$. We say ballot b is *below and incompatible than* ballot b' (written $b \lesssim b'$) iff $b < b'$ and $b \not\sim b'$. In a nutshell, ASCP works as follows: each node uses FV to *prepare* a ballot b which carries the candidate value $b.x$, this is, it aborts every ballot $b' \lesssim b$, which prevents any attempt to decide a value different from $b.x$ at a round smaller than $b.n$; once b is prepared, the node uses FV again to commit ballot b, thus deciding the candidate value $b.x$.

ASCP is depicted in Algorithm 2 above. We assume that each node v creates a process federated-voting(v, b) for each ballot b, which is stored in the infinite array ballots$[b]$ (line 2). The node keeps fields candidate and prepared, which respectively contain the ballot that v is trying to commit and the highest ballot prepared so far. Both candidate and prepared are initialised to the null ballot (line 3). The node also keeps a field round that contains the current round, initialised to 0 (line 4). Once v proposes a value x, the node assigns the ballot $\langle 1, x\rangle$ to candidate and tries to prepare it by invoking FV's primitive vote(false) for each ballot below and incompatible than candidate (lines 5–7). This may involve sending an infinite number of messages, which by BNS requires sending finitely many batches. Once v

prepares some ballot b by receiving FV's indication deliver(false) for every ballot below and incompatible than b, and if b exceeds prepared, the node updates prepared to b (lines 8–9). The condition in line 8 may concern an infinite number of ballots, but it may hold after receiving a finite number of batches by BNS. If prepared reaches or exceeds candidate, then the node updates candidate to prepared, and tries to commit it by voting *true* for that ballot (lines 10–12). Once v commits some ballot b by receiving FV's indication deliver(true) for ballot b, the node decides the value $b.x$ (lines 13–14) and stops execution.

If the candidate ballot of a node v can no longer be aborted nor committed, then v resorts to a timeout mechanism that we describe next. The primitive start-timer(Δ) starts the node's local timer, such that a timeout event will be triggered once the specified delay Δ has expired. (Invoking start-timer(Δ') while the timer is already running has the effect of restarting the timer with the new delay Δ'.) In order to start the timer, a node v needs to receive, from each member of a quorum that contains v itself, messages that endorse either committing or preparing ballots with rounds bigger than round (line 15 of Algorithm 2). Since the domain of values can be infinite, the condition in line 15 requires that for each node u in some quorum U that contains v itself, there exists a ballot b_u with round $b_u.n >$ round, and either v receives from u a message endorsing to commit b_u, or otherwise v receives from u messages endorsing to abort every ballot in some non-empty, right-open interval $[z_u, b_u)$, whose upper bound is b_u. This condition may require receiving an infinite number of ballots, but it may hold after receiving a finite number of batches by BNS. Once the condition in line 15 holds, the node updates round to the smallest n such that every member of the quorum endorses to either commit or prepare some ballot with round bigger or equal than n, and (re-)starts the timer with delay F(round), where F is an unbound function that doubles its value with each increment of n (lines 16–17). If the candidate ballot can no longer be aborted or committed, then timeout will be eventually triggered (line 18) and the node considers a new candidate ballot with the current round increased by one, and with the value candidate.x if the node never prepared any ballot yet (line 19) or the value prepared.x otherwise (line 20). Then v tries to prepare the new candidate ballot by voting *false* for each ballot below and incompatible than it (line 21). This may involve sending an infinite number of messages, which by BNS requires sending finitely many batches.

The condition for starting the timer in line 15 does not strictly use FV as a black box. However, this use is warranted because line 15 only "reads" the state of the network. ASCP makes every other change to the network through FV's primitives.

ASCP guarantees the safety properties of *non-blocking Byzantine consensus* in §3. Since a node stops execution after deciding some value, *Integrity for intact sets* holds trivially. The requirement in lines 8–12 of Algorithm 2 that a node prepares the candidate ballot before voting for committing it, enforces that if a voting for committing some ballot within the nodes of an intact set I succeeds, then some node in I previously prepared that ballot:

▶ **Lemma 9.** *Let $\mathcal{S}$ be an FBQS and consider an execution of ASCP over $\mathcal{S}$. Let I be an intact set in $\mathcal{S}$. If a node $v_1 \in I$ commits a ballot b, then some node $v_2 \in I$ prepared b.*

Aborting every ballot below and incompatible than the candidate one prevents that one node in an intact set I prepares a ballot b_1, and concurrently another node in I sends READY(b_2, *true*) with b_2 below and incompatible than b_1:

▶ **Lemma 10.** *Let $\mathcal{S}$ be an FBQS and consider an execution of ASCP over $\mathcal{S}$. Let I be an intact set in $\mathcal{S}$. Let v_1 and v_2 be nodes in I and b_1 and b_2 be ballots such that $b_2 \lesssim b_1$. The following two things cannot both happen: node v_1 prepares b_1 and node v_2 sends READY(b_2, *true*).*

Agreement for intact sets holds as follows: assume towards a contradiction that two nodes in I respectively commit ballots b_1 and b_2 with different values. A node in I prepared the bigger of the two ballots by Lemma 9, which results in a contradiction by Lemma 10.

Lemma 11 below ensures that in line 20 it is safe to take as the new candidate value that of the largest prepared ballot, which helps to establish *Weak validity for intact sets*.

▶ **Lemma 11.** *Let $\mathcal{S}$ be an FBQS and consider an execution of ASCP over $\mathcal{S}$. Let b_1 be the largest ballot prepared by some node v_1 at some moment in the execution. If all nodes are honest, then some node v_2 proposed $b_1.x$.*

Now we examine the liveness properties of *non-blocking Byzantine consensus* in §3, which ASCP also meets. Recall from §4 the bounded interval δ_I that a node in an intact set I takes to deliver some Boolean for a given ballot, provided that some other node in I has already delivered a Boolean for the same ballot. Let v be a node in I that prepares some ballot b such that no other node in I has ever prepared a ballot with round bigger or equal than $b.n$. We call the interval of duration δ_I after v prepares b the *window for intact set I of round $b.n$*. Lemma 12 below guarantees that after some moment in the execution, no two consecutive windows ever overlap.

▶ **Lemma 12.** *Let $\mathcal{S}$ be an FBQS and consider an execution of ASCP over $\mathcal{S}$. Let I be an intact set in $\mathcal{S}$ and assume that all faulty nodes eventually stop. There exists a round n such that either every node in I decides some value before reaching round n, or otherwise the windows for I of all the rounds $m \geq n$ never overlap with each other, and in each window of round m the nodes in I that have not decided yet only prepare ballots with round m.*

Lemma 12 helps to establish *Non-blocking for intact sets* as follows. After the moment where no two consecutive windows overlap, either every node in I has the same candidate ballot at the beginning of the window of some round, or otherwise the highest ballots prepared by each node in I during that window coincide with each other. In either case all the nodes in I will eventually have the same candidate ballot, and they will decide a value in bounded time.

Correctness of ASCP is captured by Theorem 13 below:

▶ **Theorem 13.** *Let $\mathcal{S}$ be an FBQS. The ASCP protocol over $\mathcal{S}$ satisfies the specification of non-blocking Byzantine consensus for intact sets.*

6 Concrete Stellar Consensus Protocol

In this section we introduce *concrete SCP* (*CSCP*) which is amenable to implementation because each node v maintains finite state and only needs to send and receive a finite number of messages in order to progress. CSCP relies on *bunched voting* (*BV*) in Algorithm 3, which generalises FV and embodies all of FV's instances for each of the ballots. CSCP considers a single instance of BV, and thus each node v keeps a single process bunched-voting(v) (line 2 of Algorithm 4). In BV, nodes exchange messages that contain two kinds of statements: a *prepare statement* PREP b encodes the aim to abort the possibly infinite range of ballots that are lower and incompatible than b; and a *commit statement* CMT b encodes the aim to commit ballot b.

Algorithm 3 depicts BV over an FBQS $\mathcal{S}$ with set of quorums $\mathcal{Q}$. A node v stores the highest ballot for which v has respectively voted, readied, or delivered a prepare statement in fields max-voted-prep, max-readied-prep, and max-delivered-prep (line 2). It also stores the set of ballots for which v has respectively voted, readied, or delivered a commit statement in fields ballots-voted-cmt, ballots-readied-cmt, and ballots-delivered-cmt (line 3). All these

Algorithm 3 Bunched voting (BV) over an FBQS $\mathcal{S}$ with set of quorums $\mathcal{Q}$.

```
process bunched-voting(v ∈ V)
    max-voted-prep, max-readied-prep, max-delivered-prep ← ⟨0, ⊥⟩ ∈ Ballot;
    ballots-voted-cmt, ballots-readied-cmt, ballots-delivered-cmt ← ∅ ∈ 2^Ballot;
    prepare(b)
        if max-voted-prep < b then
            max-voted-prep ← b;
            send VOTE(PREP max-voted-prep) to every v' ∈ V;
    when exists maximum b such that max-voted-prep < b and exists U ∈ Q
     such that v ∈ U and for every u ∈ U received VOTE(PREP b_u) where
     b' ≲ b_u for every b' ≲ b
        max-readied-prep ← b ;
        send READY(PREP max-readied-prep) to every v' ∈ V;
    when exists maximum b such that max-readied-prep < b and exists
     v-blocking B such that for every u ∈ B received READY(PREP b_u) where
     b' ≲ b_u for every b' ≲ b
        max-readied-prep ← b ;
        send READY(PREP max-readied-prep) to every v' ∈ V;
    when exists maximum b such that max-delivered-prep < b and exists U ∈ Q
     such that v ∈ U and for every u ∈ U received READY(PREP b_u) where
     b' ≲ b_u for every b' ≲ b
        max-delivered-prep ← b;
        prepared(max-delivered-prep);
    commit(b)
        if b ∉ ballots-voted-cmt and max-voted-prep = b then
            ballots-voted-cmt ← ballots-voted-cmt ∪ {b};
            send VOTE(CMT b) to every v' ∈ V ;
    when received VOTE(CMT b) from every u ∈ U for some U ∈ Q such that
     v ∈ U and b ∉ ballots-readied-cmt
        ballots-readied-cmt ← ballots-readied-cmt ∪ {b} ;
        send READY(CMT b) to every v' ∈ V ;
    when received READY(CMT b) from every u ∈ B for some v-blocking B and
     b ∉ ballots-readied-cmt
        ballots-readied-cmt ← ballots-readied-cmt ∪ {b} ;
        send READY(CMT b) to every v' ∈ V ;
    when received READY(CMT b) from every u ∈ U for some U ∈ Q such that
     v ∈ U and b ∉ ballots-delivered-cmt
        ballots-delivered-cmt ← ballots-delivered-cmt ∪ {b};
        committed(b) ;
```

fields are finite and thus v maintains only finite state. When a node v invokes $\mathsf{prepare}(b)$, if b exceeds the highest ballot for which v has voted a prepare, then the node updates $\mathsf{max\text{-}voted\text{-}prep}$ to b and sends $\texttt{VOTE}(\textsc{prep}\ b)$ to every other node (lines 4–7). The protocol then proceeds with the usual stages of FV, with the caveat that at each stage of the protocol only the maximum ballot is considered for which the node can send a message – or deliver an indication – with a prepare statement. In particular, when there exists a ballot b that exceeds $\mathsf{max\text{-}readied\text{-}prep}$ and such that v received a message $\texttt{VOTE}(\textsc{prep}\ b_u)$ from each member u of some quorum to which v belongs, then the node proceeds as follows: it checks that each b' lower and incompatible than b_u is also lower and incompatible than b (line 8). If b is the maximum ballot passing the previous check for every member u of the quorum, then the node updates the field $\mathsf{max\text{-}readied\text{-}prep}$ to b and sends the message $\texttt{READY}(\textsc{prep}\ b)$ to every other node (lines 9–10). The node v checks similar conditions for the case when it receives messages $\texttt{READY}(\textsc{prep}\ b_u)$ from each member u of a v-blocking set, and proceeds similarly by updating $\mathsf{max\text{-}readied\text{-}prep}$ to b and sending $\texttt{READY}(\textsc{prep}\ b)$ to every other node (lines 11–13). The node will update $\mathsf{max\text{-}delivered\text{-}prep}$ and trigger the indication $\mathsf{prepared}(b)$ when the same conditions are met after receiving messages $\texttt{READY}(\textsc{prep}\ b_u)$ from each member u of a quorum to which v belongs (lines 14–16). When a node v invokes $\mathsf{commit}(b)$ then the protocol proceeds with the usual stages of FV with two minor differences (lines 17–29). First, a node v only votes commit for the highest ballot for which v has voted a prepare statement (condition $\mathsf{max\text{-}voted\text{-}prep} = b$ in line 18). Second, the protocol uses the sets of ballots $\mathsf{ballots\text{-}voted\text{-}cmt}$, $\mathsf{ballots\text{-}readied\text{-}cmt}$ and $\mathsf{ballots\text{-}delivered\text{-}cmt}$ in order to keep track of the stage of the protocol for each ballot.

The structure of CSCP in Algorithm 4 directly relates to ASCP in Algorithm 2. A node proposes a value x in line 5. A node tries to prepare a ballot b by invoking $\mathsf{prepare}(b)$ in line 7, and receives the indication $\mathsf{prepared}(b)$ in line 8. A node tries to commit a ballot b by invoking $\mathsf{commit}(b)$ in line 12, and receives the indication $\mathsf{committed}(b)$ in line 13. A node decides a value x in line 14. Timeouts are set in lines 15–17 and triggered in line 18.

Next we establish a correspondence between CSCP in and ASCP in §5: the concrete protocol observationally refines the abstract one, which means that any externally observable behaviour of the former can also be produced by the latter [6]. Informally, the refinement shows that for an FBQS $\mathcal{S}$ and an intact set I, for every execution of CSCP over $\mathcal{S}$ there exists an execution of ASCP over $\mathcal{S}$ (with some behaviour of faulty nodes) such that each node in I decides the same value in both of the executions. The refinement result allows us to carry over the correctness of ASCP established in §5 to CSCP.

We first define several notions required to formalise our refinement result. A *history* is a sequence of the events $v.\mathsf{propose}(x)$ and $v.\mathsf{decide}(x)$, where v is a correct node and x a value. The specification of consensus assumes that v triggers an event $v.\mathsf{propose}(x)$, thus a history will have $v.\mathsf{propose}(x)$ for every correct node v. A *concrete trace* τ is a sequence of events that subsumes histories, and contains events $v.\mathsf{prepare}(b)$, $v.\mathsf{commit}(b)$, $v.\mathsf{prepared}(b)$, $v.\mathsf{committed}(b)$, $v.\mathsf{start\text{-}timer}(n)$, $v.\mathsf{timeout}$, $v.\mathsf{send}(m, v')$, and $v.\mathsf{receive}(m, v')$, where v is a correct node and v' is any node, b is a ballot, m is a message in $\{\texttt{VOTE}(s), \texttt{READY}(s)\}$ with s a statement in $\{\textsc{prep}\ b, \textsc{cmt}\ b\}$, and n is a round. An *abstract trace* τ is a sequence of events that subsumes histories, and contains events $v.\mathsf{start\text{-}timer}(n)$, $v.\mathsf{timeout}$, and batched events $v.\mathsf{vote\text{-}batch}([b_i], a)$, $v.\mathsf{deliver\text{-}batch}([b_i], a)$, $v.\mathsf{send\text{-}batch}([m_i], v')$, and $v.\mathsf{receive\text{-}batch}([m_i], v')$,where v is a correct node and v' is any node, n is a round, $[b_i]$ is a sequence of ballots, a is a Boolean, and $[m_i]$ is a sequence of messages in $\{\texttt{VOTE}(b, a), \texttt{READY}(b, a)\}$. The sequences of ballots and messages above, which represent a possibly infinite number of "batched" events, ensure that the length of any abstract trace is bounded by ω. We may

Algorithm 4 Concrete SCP (CSCP) over an FBQS $\mathcal{S}$ with set of quorums $\mathcal{Q}$.

```
1  process concrete-consensus(v ∈ V)
2      brs ← new process bunched-voting(v);
3      candidate, prepared ← ⟨0, ⊥⟩ ∈ Ballot;
4      round ← 0 ∈ ℕ⁺ ∪ {0};
5      propose(x)
6          candidate ← ⟨1, x⟩;
7          brs.prepare(candidate);
8      when triggered brs.prepared(b) and prepared < b
9          prepared ← b;
10         if candidate ≤ prepared then
11             candidate ← prepared;
12             brs.commit(candidate);
13     when triggered brs.committed(b)
14         trigger decide(b.x);
15     when exists U ∈ Q such that v ∈ U and for each u ∈ U exist
       M_u ∈ {VOTE, READY} and b_u ∈ Ballot such that round < b_u.n and received
       M_u(S_u b_u) from u with S_u ∈ {CMT, PREP}
16         round ← min{b_u.n | u ∈ U};
17         start-timer(F(round));
18     when triggered timeout
19         if prepared = ⟨0, ⊥⟩ then candidate ← ⟨round + 1, candidate.x⟩;
20         else candidate ← ⟨round + 1, prepared.x⟩;
21         brs.prepare(candidate);
```

omit the adjective "concrete/abstract" from "trace" when it is clear from the context. Given a trace τ, a history $H(\tau)$ can be uniquely obtained from τ by removing every event in τ different from $v.\mathsf{propose}(x)$ or $v.\mathsf{decide}(x)$.

An execution of CSCP (respectively, ASCP) *entails* a *concrete trace* (respectively, *abstract trace*) τ iff for every invocation and indication as well as for every send or receive primitive in an execution of the protocol in Algorithm 4 (respectively, for every invocation, indication and primitive in an execution of the protocol in Algorithm 2, where the vote, deliver, send and receive events are batched together), τ contains corresponding events in the same order.

We are interested in traces that are relative to some intact set I. Given a trace τ, the *I-projected* trace $\tau|_I$ is obtained by removing the events $v.\mathsf{ev} \in \tau$ such that $v \notin I$.

▶ **Theorem 14.** *Let $\mathcal{S}$ be an FBQS and I be an intact set. For every execution of CSCP over $\mathcal{S}$ with trace τ, there exists an execution of ASCP over $\mathcal{S}$ with trace ρ and $H(\tau|_I) = H(\rho|_I)$.*

Proof sketch. We define a *simulation function* σ from concrete to abstract traces. Theorem 14 can be established by showing that, for every finite prefix τ of a trace entailed by CSCP, the simulation $\sigma(\tau)$ is a prefix of a trace entailed by ASCP. ◀

Every execution of ASCP enjoys the properties of *Integrity*, *Agreement for intact sets*, *Weak validity for intact sets* and *Non-blocking for intact sets*, and so does every execution of CSCP by refinement.

▶ **Corollary 15.** *Let $\mathcal{S}$ be an FBQS. The CSCP protocol over $\mathcal{S}$ satisfies the specification of non-blocking Byzantine consensus for intact sets.*

7 Lying about Quorum Slices

So far we have assumed the unrealistic setting where faulty nodes do not equivocate their quorum slices, so all nodes share the same FBQS $\mathcal{S}$. We now lift this assumption. To this end, we use a generalisation of FBQS called *subjective FBQS* [7], which allows faulty nodes to lie about their quorum slices. Assuming that $\mathbf{V}_{\text{ok}}$ is the set of correct nodes, the *subjective FBQS* $\{\mathcal{S}_v\}_{v\in\mathbf{V}_{\text{ok}}}$ is an indexed family of FBQSes where the different FBQSes agree on the quorum slices of correct nodes, *i.e.*, $\forall v_1, v_2, v \in \mathbf{V}_{\text{ok}}.\ \mathcal{S}_{v_1}(v) = \mathcal{S}_{v_2}(v)$. For each correct node v, the FBQS $\mathcal{S}_v$ is the *view* of node v, which reflects the choices of trust communicated to v. We can run either ASCP or CSCP over a subjective FBQS $\{\mathcal{S}_v\}_{v\in\mathbf{V}_{\text{ok}}}$ by letting each correct node v act according to its view $\mathcal{S}_v$.

We generalise the definition of intact set to subjective FBQSes, and we lift our results so far to the subjective FBQSes. Let $\{\mathcal{S}_v\}_{v\in\mathbf{V}_{\text{ok}}}$ be a subjective FBQS. A set I is an *intact set* iff for each $v \in \mathbf{V}_{\text{ok}}$ the set I is a quorum in $\mathcal{S}_v$ that only contains correct nodes, and every member of I is intertwined with each other in the projected FBQS $\mathcal{S}_v|_I$.

▶ **Lemma 16.** *Let $\{\mathcal{S}_v\}_{v\in\mathbf{V}_{\text{ok}}}$ be a subjective FBQS. For any node $v \in \mathbf{V}_{\text{ok}}$, a set I is an intact set in $\mathcal{S}_v$ iff I is an intact set in $\{\mathcal{S}_v\}_{v\in\mathbf{V}_{\text{ok}}}$.*

Since Lemma 16 above guarantees that every view has the same intact sets, which also coincide with the intact sets of the subjective FBQS, from now on we may say "an intact set I" and omit to which system (a particular view, or the subjective FBQS) I belongs .

Using the fact that nodes agree on the slices of correct nodes, we can prove Lemma 17 below, which is the analogue to Lemma 3 and states sufficient safety conditions for the nodes in an intact set I to reach agreement when each node acts according to its own view.

▶ **Lemma 17.** *Let $\{\mathcal{S}_v\}_{v\in\mathbf{V}_{\text{ok}}}$ be a subjective FBQS and for each correct node v let $\mathcal{Q}_v$ be the set of quorums in the view $\mathcal{S}_v$. Let I be an intact set and consider two quorums U_1 and U_2 in $\bigcup_{v\in\mathbf{V}_{\text{ok}}} \mathcal{Q}_v$. If $U_1 \cap I \neq \emptyset$ and $U_2 \cap I \neq \emptyset$, then $U_1 \cap U_2 \cap I \neq \emptyset$.*

Using arguments similar to those in the previous sections, we can establish the correctness of ASCP and CSCP over subjective FBQSes.

▶ **Theorem 18.** *Let $\{\mathcal{S}_v\}_{v\in\mathbf{V}_{\text{ok}}}$ be a subjective FBQS. The ASCP protocol over $\{\mathcal{S}_v\}_{v\in\mathbf{V}_{\text{ok}}}$ satisfies the specification of non-blocking Byzantine consensus for intact sets.*

▶ **Theorem 19.** *Let $\{\mathcal{S}_v\}_{v\in\mathbf{V}_{\text{ok}}}$ be a subjective FBQS and I be an intact set. For every execution of CSCP over $\{\mathcal{S}_v\}_{v\in\mathbf{V}_{\text{ok}}}$ with trace τ, there exists an execution of ASCP over $\{\mathcal{S}_v\}_{v\in\mathbf{V}_{\text{ok}}}$ with trace ρ and $H(\tau|_I) = H(\rho|_I)$.*

▶ **Corollary 20.** *Let $\{\mathcal{S}_v\}_{v\in\mathbf{V}_{\text{ok}}}$ be a subjective FBQS. The CSCP protocol over $\{\mathcal{S}_v\}_{v\in\mathbf{V}_{\text{ok}}}$ satisfies the specification of non-blocking Byzantine consensus for intact sets.*

8 Related Work

García-Pérez and Gotsman [7] have previously investigated Stellar's federated voting and its relationship to Bracha's broadcast over classical Byzantine quorum systems. They did not address the full Stellar consensus protocol. Our proof of SCP establishes the correctness of federated voting by adjusting the results in [7] to multiple intact sets within the system.

Losa et al. [10] have also investigated consensus over FBQSs. They propose a generalisation of Stellar's quorums that does not prescribe constructing them from slices, yet allows different participants to disagree on what constitutes a quorum. They then propose a protocol solving consensus over intact sets in this setting that provides better liveness guarantees than SCP, but is impractical. Losa et al.'s work is orthogonal to ours: they consider a more general setting than Stellar's and a theoretical protocol, whereas we investigate the practical protocol used by Stellar.

The advent of blockchain has given rise to a number of novel proposals of BFT protocols; see [3] for a survey. Out of these, the most similar one to Stellar is Ripple [15]. In particular, Ripple have recently proposed a protocol called Cobalt that allows for a federated setting similar to Stellar's [11]. We hope that our work will pave the way to investigating the correctness of this and similar protocols.

References

1. Gabriel Bracha. Asynchronous Byzantine Agreement Protocols. *Information and Computation*, 75(2):130–143, 1987.
2. Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.
3. Christian Cachin and Marko Vukolic. Blockchain Consensus Protocols in the Wild. In *International Symposium on Distributed Computing (DISC)*, 2017.
4. Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, 2002.
5. Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, and Bastian Pochon. The perfectly synchronized round-based model of distributed computing. *Information and Computation*, 205(5):783–815, 2007.
6. Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzky, and Hongseok Yang. Abstraction for concurrent objects. *Theoretical Computer Science*, 411(51-52):4379–4398, 2010.
7. Álvaro García-Pérez and Alexey Gotsman. Federated Byzantine Quorum Systems. In *International Conference on Principles of Distributed Systems (OPODIS)*, volume 125 of *LIPIcs*, pages 17:1–17:16. Schloss Dagstuhl, 2018.
8. Álvaro García-Pérez and Maria A. Schet. Deconstructing Stellar Consensus (extended version). *CoRR*, abs/1911.05145, 2019. URL: `http://arxiv.org/abs/1911.05145`.
9. Leslie Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.
10. Giuliano Losa, Eli Gafni, and David Mazières. Stellar consensus by instantiation. In *33rd International Symposium on Distributed Computing (DISC 2019)*, volume 146 of *LIPIcs*, pages 27:1–27:15. Schloss Dagstuhl, 2019.
11. Ethan MacBrough. Cobalt: BFT Governance in Open Networks. *CoRR*, abs/1802.07240, 2018. URL: `http://arxiv.org/abs/1802.07240`.
12. Dahlia Malkhi and Michael K. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.
13. David Mazières. The Stellar consensus protocol: a federated model for internet-Level consensus, 2015. URL: `https://www.stellar.org/papers/stellar-consensus-protocol.pdf`.
14. David Mazières, Giuliano Losa, and Eli Gafni. Simplifed SCP, 2019. URL: `http://www.scs.stanford.edu/~dm/blog/simplified-scp.html`.
15. David Schwartz, Noah Youngs, and Arthur Britto. The Ripple Protocol Consensus Algorithm, 2014. URL: `https://ripple.com/files/ripple_consensus_whitepaper.pdf`.
16. Yee Jiun Song, Robbert van Renesse, Fred B. Schneider, and Danny Dolev. The Building Blocks of Consensus. In *International Conference on Distributed Computing and Networking (ICDCN)*, 2008.

Byzantine-Tolerant Set-Constrained Delivery Broadcast

Alex Auvolat
École Normale Supérieure, Paris, France
Univ Rennes, Inria, CNRS, IRISA, Rennes, France
https://team.inria.fr/wide/team/alex-auvolat-bernstein/
alex.auvolat@inria.fr

Michel Raynal
Univ Rennes, Inria, CNRS, IRISA, Rennes, France
Department of Computing, Polytechnic University, Hong Kong
https://www.irisa.fr/prive/raynal/
michel.raynal@irisa.fr

François Taïani
Univ Rennes, Inria, CNRS, IRISA, Rennes, France
https://ftaiani.ouvaton.org/
francois.taiani@irisa.fr

Abstract

Set-Constrained Delivery Broadcast (SCD-broadcast), recently introduced at ICDCN 2018, is a high-level communication abstraction that captures ordering properties not between individual messages but between sets of messages. More precisely, it allows processes to broadcast messages and deliver sets of messages, under the constraint that if a process delivers a set containing a message m before a set containing a message m', then no other process delivers first a set containing m' and later a set containing m. It has been shown that SCD-broadcast and read/write registers are computationally equivalent, and an algorithm implementing SCD-broadcast is known in the context of asynchronous message passing systems prone to crash failures.

This paper introduces a Byzantine-tolerant SCD-broadcast algorithm, which we call BSCD-broadcast. Our proposed algorithm assumes an underlying basic Byzantine-tolerant reliable broadcast abstraction. We first introduce an intermediary communication primitive, Byzantine FIFO broadcast (BFIFO-broadcast), which we then use as a primitive in our final BSCD-broadcast algorithm. Unlike the original SCD-broadcast algorithm that is tolerant to up to $t < n/2$ crashing processes, and unlike the underlying Byzantine reliable broadcast primitive that is tolerant to up to $t < n/3$ Byzantine processes, our BSCD-broadcast algorithm is tolerant to up to $t < n/4$ Byzantine processes. As an illustration of the high abstraction power provided by the BSCD-broadcast primitive, we show that it can be used to implement a Byzantine-tolerant read/write snapshot object in an extremely simple way.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Software and its engineering → Abstraction, modeling and modularity

Keywords and phrases Algorithm, Asynchronous system, Byzantine process, Communication abstraction, Distributed computing, Distributed software engineering, Fault-tolerance, Message-passing, Modularity, Read/write snapshot object, Reliable broadcast, Set-constrained message delivery

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.6

Acknowledgements This work has been partially supported by the French ANR projects DESCARTES n. ANR-16-CE40-0023, and PAMELA n. ANR-16-CE23-0016.

1 Introduction

Reliable broadcast in asynchronous crash-prone systems

Reliable broadcast is a communication abstraction central to fault-tolerant asynchronous distributed systems. It allows each process to broadcast, with well-defined delivery guarantees,

© Alex Auvolat, Michel Raynal, and François Taïani;
licensed under Creative Commons License CC-BY
23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller; Article No. 6; pp. 6:1–6:23
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

messages to all processes in the presence of failures. More precisely, it guarantees that non-faulty processes deliver the same set of messages M, including all the messages they broadcast, and a subset of the messages broadcast by faulty processes before they crashed. The fundamental property of the reliable broadcast abstraction asserts that no two non-faulty processes deliver different sets of messages, and a faulty process delivers a subset of the messages delivered by the non-faulty processes (two distinct faulty processes possibly delivering different sets of messages).

Reliable broadcast in the presence of Byzantine processes (BR-broadcast)

Reliable broadcast has been studied in the context of Byzantine failures since the eighties. A process commits a Byzantine failure if it behaves arbitrarily (i.e., its behavior is not the one described by the algorithm is it assumed to execute) [10, 12]. Such a failure can be intentional (also called malicious) or the result of a transient fault which altered its intended behavior in an unpredictable way. Bracha [3] introduced an elegant signature-free Byzantine fault-tolerant algorithm for the reliable broadcast abstraction in n-process message-passing asynchronous systems, where up to $t < n/3$ processes may be Byzantine.

Set-Constrained Delivery broadcast in asynchronous crash-prone systems

Set-Constrained Delivery broadcast (SCD-broadcast) was introduced in [8] in the context of crash failures. Rather than individual messages, a process delivers *non-empty sets* of messages, satisfying the following ordering property: if a non-faulty process delivers a set of message $ms1$ containing a message m and later delivers a set of message $ms1'$ containing a message m', no non-faulty process delivers first a set of message $ms2$ containing m' and later a set of message $ms2'$ containing m. This communication abstraction is particularly efficient to build read/write implementable objects, such as the ones described in [14]. It is shown in [8] that, in asynchronous message-passing systems with up to t crashed processes, (a) $t < n/2$ is necessary and sufficient to build SCD-broadcast, and (b) atomic R/W registers and SCD-broadcast have the same computability power.

Content of the paper

In this paper, we introduce a new communication abstraction, which we call BSCD-broadcast, that provides similar guarantees to SCD broadcast in the context of Byzantine failures. The specification and implementation of such a high-level abstraction has a direct practical interest when realizing many replicated objects. As an example, we show how a Byzantine-tolerant single-writer/multi-reader snapshot object [1] can be easily built on top of BSCD-broadcast. More generally, we believe a better understanding of how high-level broadcast abstractions can be implemented in a Byzantine context can help developers design novel and richer zero-trust applications, extending their use and applicability beyond the highly-publicized examples of cryptocurrencies [5, 11] and smart contract platforms [4].

We begin by defining a first abstraction (Section 3.2), BFIFO-broadcast, that guarantees that non-faulty processes deliver messages from each process in their sending order. Since such a sending order cannot be defined for Byzantine processes, the only guarantee provided in that case is that all non-faulty processes deliver the messages from a given Byzantine process in the same order. Depending on the BR-broadcast algorithm used (e.g., [3, 9]), we obtain for BFIFO-broadcast either an algorithm which requires (i) $t < n/3$ and three sequential communication steps, or (ii) $t < n/5$ and two sequential communication steps.

This simple abstraction is then used to build a BSCD-broadcast algorithm (Section 3) under the assumption $t < n/4$. The design of our algorithm differs significantly from the

SCD-broadcast algorithm described in [8], due to the fact that in the crash-failure model, a process behaves correctly until it possibly crashes, whereas a Byzantine process can exhibit an arbitrary behavior at any time. Precise definitions of both of these communication abstractions are provided in the paper. Our BSCD-broadcast algorithm requires two sequential BFIFO-broadcast steps for each BSCD-broadcast message. These two algorithms are built upon a signature-free Byzantine reliable broadcast primitive and are equally signature-free.

Finally, we build a Byzantine-tolerant snapshot object as an example application of BSCD-broadcast (Section 4). Let us notice that it has recently been shown that the snapshot object is instrumental in the implementation of cryptocurrencies [5]. It follows that the simple implementation (presented below) of such an object in message-passing systems where processes can exhibit Byzantine failures can benefit to cryptocurrencies and some other blockchain-based applications.

2 Computation Model

2.1 On the process side

Asynchronous processes

The system is made up of a finite set Π of $n > 1$ asynchronous sequential processes, namely $\Pi = \{p_1, \ldots, p_n\}$. *Asynchronous* means that each process proceeds at its own speed, which can vary arbitrarily with time, and always remains unknown to the other processes.

Process failures

Up to t processes can exhibit a *Byzantine* behavior. 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. As a simple example, a Byzantine process, which is assumed to broadcast a message m to all the processes, can send a message m_1 to some processes, a different message m_2 to another subset of processes, and no message at all to the remaining processes. Moreover, Byzantine processes can collude to foil non-Byzantine processes. It is however assumed that a Byzantine process cannot send an infinite number of messages in a finite time, a necessary hypothesis in our proof of termination. A process that exhibits a Byzantine behavior is also called *faulty*. Otherwise, it is *correct* or *non-faulty*.

2.2 On the communication side

The basic Byzantine reliable broadcast communication abstraction

This abstraction, denoted BR-broadcast, is a one-shot communication abstraction that provides two operations, br_broadcast() and br_deliver(). "One-shot" means that a process executes br_broadcast() at most once, and br_deliver() at most n times (one per possible sender). As in [6, 13], we use the following terminology: when a process invokes br_broadcast(), we say that it "br-broadcasts a message", and when it executes br_deliver(), we say that it "br-delivers a message". BR-broadcast is defined by the following properties:

- BR-Validity. If a correct process br-delivers a message m from a correct process p_i, then p_i br-broadcast m.
- BR-Integrity. A correct process br-delivers at most one message m from a process p_i.
- BR-Termination-1. If a correct process br-broadcasts a message, it br-delivers it.
- BR-Termination-2. If a correct process br-delivers a message m from p_i (possibly faulty) then all correct processes eventually br-deliver m from p_i.

On the safety side, BR-validity relates the outputs (messages br-delivered) to the inputs (messages br-broadcast), while BR-integrity states that there is no duplication.

On the liveness side, BR-Termination-2 gives its name to reliable broadcast: be the sender correct or not, every message br-delivered by a correct process is br-delivered by all correct processes. Coupled together, the two termination properties further imply that a message br-broadcast by a correct process is br-delivered by all correct processes. It follows from these properties that all correct processes br-deliver the same set of messages, and this set contains at least all the messages br-broadcast by correct processes.

As indicated in the introduction, there are signature-free distributed algorithms, which build BR-broadcast on top of asynchronous message-passing systems in which processes may be Byzantine [3, 9].

Terminology

When studying the BR-broadcast abstraction, a message m br-broadcast by a process is called an *application* message. Differently, a message generated by the algorithm implementing the abstraction is called a *protocol* message. Similarly in the rest of this paper when studying other broadcast abstractions, messages handled by that abstraction are referred to as application messages, and messages generated by the algorithm implementing the abstraction (possibly through a lower-level abstraction) are referred to as protocol messages.

2.3 Multi-shot Byzantine reliable broadcast

As already stated, BR-broadcast is a one-shot communication abstraction. But BR-broadcast allows several processes to invoke the operation br_broadcast(), each giving rise to distinct BR-broadcast instances. Two BR-broadcast instances can be easily distinguished by associating with each of them the identity of the process that created it.

It follows that a multi-shot BR-broadcast abstraction, which we call MBR-broadcast, can be very easily obtained by adding a sequence number sn to each BR-broadcast instance. A MBR-broadcast instance is then identified by a pair $\langle i, sn\rangle$, but only sn needs to be provided in an invocation, namely p_i must invoke br_broadcast(sn_i, m), where sn_i is the local integer variable (initialized to 0) used by p_i to generate its sequence numbers. Conversely, when a message is MBR-delivered by a process p_i, the information provided to the upper layer is a triple $\langle j, sn, m\rangle$ and we say that p_i mbr-delivered the message m from p_j with sequence number sn.

The BR-Validity and BR-Integrity properties for this new multi-shot abstraction become:

- MBR-Validity. If a correct process mbr-delivers a message m from a correct process p_i with sequence number sn, then p_i mbr-broadcast m with sequence number sn.
- MBR-Integrity. Given a sequence number sn, a correct process mbr-delivers at most one message m associated with sn from a process p_i.

The other properties MBR-Termination-1 and MBR-Termination-2 remain the same as for BR-broadcast. Multi-shot extensions of the single-shot signature-free BR-broadcast algorithms introduced in [3, 9] are presented in Appendix A.

3 Byzantine Set-Constrained Delivery Broadcast

3.1 Definition

Set-Constrained Delivery broadcast (SCD-broadcast) was introduced in [8], in the context of asynchronous systems prone to process crashes (in which it can be built if and only if $t < n/2$).

We consider here its extension to Byzantine process failures, denoted BSCD-broadcast. This communication abstraction provides two operations, bscd_broadcast() and bscd_deliver(). (We say that a process bscd-broadcasts messages and bscd-delivers messages.) The operation bscd_broadcast(m) allows the invoking process to broadcast an application message, while the operation bscd_deliver() returns a non-empty set of messages to the invoking process. BSCD-broadcast is defined by the following properties.

- BSCD-Validity. If a correct process bscd-delivers a set of messages containing a message m from a process p_i, if p_i is correct, it bscd-broadcast m.
- BSCD-Integrity. A message is bscd-delivered at most once by each correct process.
- BSCD-Order. Let p_i be a correct process that first bscd-delivers a set of messages ms_i and later bscd-delivers a set of messages ms'_i. For any pair of messages $m \in ms_i$ and $m' \in ms'_i$, no correct process bscd-delivers first a set containing m' and later bscd-delivers a set containing m.
- BSCD-Termination-1. If a correct process bscd-broadcasts a message m, it bscd-delivers a message set containing m.
- BSCD-Termination-2. If a correct process bscd-delivers a message set containing m, every correct process bscd-delivers a message set containing m.

As a simple example, let m_1, m_2, m_3, m_4, m_5, m_6, m_7 and m_8 be messages that have been bscd-broadcast by different processes. The following message set bscd-deliveries by p_1, p_2 and p_3 respect the definition of BSCD-broadcast:

- at p_1: $\{m_1, m_2\}$, $\{m_3, m_4, m_5\}$, $\{m_6\}$, $\{m_7, m_8\}$.
- at p_2: $\{m_1\}$, $\{m_3, m_2\}$, $\{m_6, m_4, m_5\}$, $\{m_7\}$, $\{m_8\}$.
- at p_3: $\{m_3, m_1, m_2\}$, $\{m_6, m_4, m_5\}$, $\{m_7\}$, $\{m_8\}$.

Differently, due to the deliveries of the message sets including m_2 and m_3, the following message set deliveries by p_1 and p_2 do not satisfy the BSCD-Order property.

- at p_1: $\{m_1, m_2\}$, $\{m_3, m_4, m_5\}$, ...
- at p_2: $\{m_1, m_3\}$, $\{m_2\}$, ...

3.2 A simple sub-protocol to ease presentation: BFIFO Broadcast

In order to simplify the presentation of our BSCD-broadcast algorithm, we first introduce a straightforward multi-shot first-in-first-out Byzantine broadcast primitive (BFIFO-broadcast), that is implemented on top of MBR-broadcast (Section 2.3) by Algorithm 1. Basing our algorithm on this abstraction allows us to assume that all correct processes agree on the order of relevant messages for any possible sender, *even a Byzantine one*. While ordering messages from each possible sender is an important first step, the crucial property of introducing order between messages of different processes is achieved later, in Algorithm 2.

BFIFO-broadcast is defined by the four MBR-broadcast properties (renamed BFIFO-Validity, BFIFO-Integrity, BFIFO-Termination-1, and BFIFO-Termination-2), to which we add a FIFO delivery guarantee, defined as follows:

- BFIFO-Order. If a correct process p_i bfifo-delivers two messages m and m' from the same process p_k in the order first m and then m', no correct process bfifo-delivers m' before m (BFIFO-Order-1). Moreover, if p_k is correct, it bfifo-broadcast m before m' (BFIFO-Order-2).

In practice, each invocation of bfifo_broadcast() by a process p_i is identified by the pair $\langle i, sn \rangle$, where sn is the corresponding sequence number. The bfifo-delivery order at correct processes corresponds to the order of increasing sequence numbers.

The proof that Algorithm *1* implements BFIFO-broadcast is provided in Appendix B.1.

Algorithm 1 BFIFO-broadcast on top of MBR-broadcast (code for p_i).

init $sn_i \leftarrow 0$; $fifo_del_i \leftarrow [0, \ldots, 0]$.

operation bfifo_broadcast(m) **at** p_i **is**
(1) $sn_i \leftarrow sn_i + 1$;
(2) br_broadcast(sn_i, m).

when $\langle j, sn, m\rangle$ **is** br_delivered **at** p_i **do**
(3) wait$\big(sn = fifo_del_i[j] + 1\big)$;
(4) bfifo_deliver $\langle j, sn, m\rangle$;
(5) $fifo_del_i[j] \leftarrow fifo_del_i[j] + 1$.

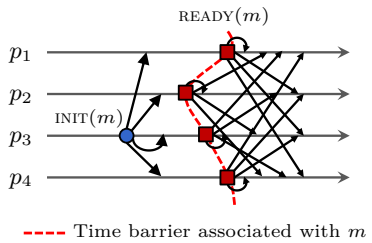


Figure 1 The echo mechanism of Algorithm 2.

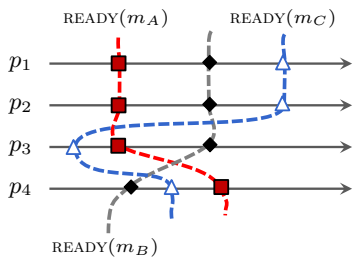


Figure 2 Three time barriers.

3.3 An algorithm for BSCD-broadcast on top of BFIFO-broadcast

Algorithm 2 implements BSCD-broadcast on top of BFIFO-broadcast, with the assumption $t < n/4$. This assumption is required in our proof of correctness (see proof of Lemma 17). An open question remains as to whether $t < n/4$ is a tight bound for BSCD-broadcast.

This algorithm is used in Section 4 to build a Byzantine-tolerant snapshot object.

Overall intuition and challenges

At its core, Algorithm 2 must prevent any potential disruption caused by Byzantine processes, as these cannot be assumed to respect any given behaviour. This need for containment of impredictible behaviour leads to a design that departs fundamentally from the crash-tolerant SCD-broadcast algorithm proposed in [8].

Whereas the crash-tolerant version SCD-broadcast was able to inject order into the system by enforcing a waiting period between successive broadcasts from the same source, this strategy no longer works in a Byzantine setting. This is because nothing prevents Byzantine processes from issuing overlapping broadcasts, in order to confuse correct processes and foil the protocol. This apparently subtle limitation renders the protocol considerably more complex, as correct processes must now cooperate to enforce an order on the broadcasts initiated by a possibly Byzantine source, while pruning inconsistent control information produced by Byzantine processes.

More concretely, Algorithm 2 exploits an echo mechanism (the READY messages bfifo-broadcast at lines 5 and 23, illustrated in Figure 1) to construct temporal barriers that witness a message's distribution among participants. The construction of these barriers is however constrained by preventing correct processes from contributing to the barrier of a bscd-broadcast from a source p_i, if an earlier bscd-broadcast by the same p_i (*earlier* in the sense of the bfifo-broadcast) has not yet been bscd-delivered. This sequencing introduces some order into the set of bscd-broadcast messages unfolding concurrently, and eventually

Algorithm 2 BSCD-broadcast on top of BFIFO-broadcast ($t < n/4$, code for p_i).

init $pending_i \leftarrow [\emptyset, \ldots, \emptyset]$;
  for each $j \in \{1, \cdots, n\}$, $snj \geq 1$
    do $data_i[j, snj] \leftarrow \perp$; $witness_i[j, snj] \leftarrow [+\infty, \cdots, +\infty]$
  end for.

operation bscd_broadcast(m) **at** p_i **is**
(1) bfifo_broadcast INIT(m).

when $\langle j, snj, \text{INIT}(m)\rangle$ **is** bfifo_delivered **at** p_i
(2) % snj is the sequence number carried by INIT(m); namely: $fifo_del_i[j] = snj$ (cf. Alg. 1)
(3) $data_i[j, snj] \leftarrow m$; % we then have $\mathsf{id}(m) = \langle j, snj\rangle$
(4) $pending_i[j] \leftarrow pending_i[j] \cup \{snj\}$;
(5) **if** $snj = \min(pending_i[j])$ **then** bfifo_broadcast READY($\langle j, snj\rangle$) **end if**;
(6) try_deliver().

when $\langle k, snk, \text{READY}(\langle j, snj\rangle)\rangle$ **is** bfifo_delivered **at** p_i
(7) % snk is the sequence number carried by READY($\langle j, snj\rangle$); namely: $fifo_del_i[k] = snk$
(8) **if** $witness_i[j, snj][k] = +\infty$ **then** $witness_i[j, snj][k] \leftarrow snk$ **end if**;
(9) **if** $snj > fifo_del_i[j] \ \wedge \ |\{k'$ such that $witness_i[j, snj][k'] < +\infty\}| \geq t + 1$
(10) **then** $pending_i[j] \leftarrow pending_i[j] \cup \{snj\}$ **end if**;
(11) try_deliver().

internal operation try_deliver () **is**
(12) $candidates_i \leftarrow \{\langle j, snj\rangle$ such that
$(snj \in pending_i[j]) \wedge (|\{k$ such that $witness_i[j, snj][k] < +\infty\}| \geq t + 1)\}$;
(13) $todel_i \leftarrow \{\langle j, snj\rangle \in candidates_i$ such that
$(data_i[j, snj] \neq \perp) \wedge (|\{k$ such that $witness_i[j, snj][k] < +\infty\}| \geq n - t)\}$;
(14) **while** $\exists \langle j, snj\rangle \in todel_i : \neg \big(\forall \langle j', snj'\rangle \in candidates_i \setminus todel_i, \mathsf{safe}(\langle j, snj\rangle, \langle j', snj'\rangle)\big)$
(15) **do** $todel_i \leftarrow todel_i \setminus \{\langle j, snj\rangle\}$
(16) **end while**;
(17) **if** $todel_i \neq \emptyset$ **then**
(18) $ms \leftarrow \{\langle j, snj, data_i[j, snj]\rangle$ such that $\langle j, snj\rangle \in todel_i\}$;
(19) bscd_deliver(ms);
(20) **for each** $\langle j, snj\rangle \in todel_i$ **in increasing lexicographical order do**
(21) $pending_i[j] \leftarrow pending_i[j] \setminus \{snj\}$;
(22) **if** $(pending_i[j] \neq \emptyset) \wedge (data_i[j, \min(pending_i[j])] \neq \perp)$
(23) **then** bfifo_broadcast READY($\langle j, \min(pending_i[j])\rangle$) **end if**
(24) **end for**
(25) **end if**.

internal predicate safe ($\langle j, snj\rangle, \langle j', snj'\rangle$) **is**
(26) **if** $(j = j' \wedge snj < snj')$ **then** return (`true`) **end if**;
(27) **if** $|\{k$ such that $witness_i[j, snj][k] < witness_i[j', snj'][k]\}| > \frac{n}{2}$ **then** return (`true`) **end if**;
(28) return (`false`).

ensures that received bscd-broadcasts can be locally bundled together and delivered as sets respecting the BSCD-broadcast specification by correct processes.

A note on sequence numbers and message identity

Application messages, each corresponding to an invocation of bscd_broadcast(), are identified by a pair $\langle j, snj\rangle$ made of a process identity j and a sequence number snj. To simplify the presentation of the algorithm, and without loss of generality, the sequence numbers given to bscd-broadcast application messages correspond to the sequence number of an underlying bfifo-broadcast message noted INIT(m) in the algorithm. However the BSCD-broadcast algorithm also bfifo-broadcasts protocol messages of other types (namely, READY messages), therefore many sequence numbers of bfifo-broadcast messages do not correspond to a bscd-broadcast message. As a consequence, these application messages do not have strictly sequential numbers.

Local variables at a process

Each process p_i manages the following local variables:

- $pending_i[1..n]$ is an array of sets, such that $pending_i[j]$ contains the sequence numbers of the messages known by p_i as bscd-broadcast by p_j and not yet locally bscd-delivered.
- $data_i[1..n, 1...]$ is a two-dimensional array, whose entries are initially $\perp$. $data_i[j, snj]$ stores the content of message $\langle j, snj\rangle$ (the message from p_j whose sequence number is snj).
- $witness_i[1..n, 1...]$ is another two-dimensional array that records the temporal barriers observed so far by p_i. Each entry $witness_i[j, snj]$ stores the barrier for message $\langle j, snj\rangle$, an array of size n initialized to $[+\infty, \cdots, +\infty]$. As the algorithm progresses, $witness_i[j, snj][k]$ is updated to contain the logical date snk (a sequence number of p_k) at which p_k witnessed message $\langle j, snj\rangle$ by bfifo-broadcasting the protocol message READY($\langle j, snj\rangle$).
- $candidates_i$ and $todel_i$ are sets containing the identities of message which are potential candidates to belong to the next message set bscd-delivered by p_i; $todel_i$ is a refined subset of $candidates_i$.

Behavior of a process p_i

A process triggers the bscd-broadcast of a message m by simply bfifo-broadcasting the protocol message INIT(m) (line 1, illustrated in Fig. 1).

When a process p_i bfifo-delivers a message INIT(m) from a process p_j, this protocol message has been allocated a sequence number called snj (line 2) by the underlying BFIFO broadcast algorithm (Section 3.2), and consequently $\langle j, snj\rangle$ becomes the identity of the application message m. m is first stored in $data_i[j, snj]$ (line 3), and its sequence number snj is then added to $pending_i[j]$ (line 4), to record that p_i knows that p_j used an INIT message to trigger a bscd-broadcast for the message with sequence number snj, and that p_i must bscd-deliver this message. Finally, if snj is the smallest sequence number in $pending_i[j]$, p_i knows that all earlier messages from p_j have been bscd-delivered, and that $\langle j, snj\rangle$ is the next message it will bscd-deliver from p_j. It informs of it the other processes by bfifo-broadcasting the protocol message READY($\langle j, snj\rangle$) (line 5, illustrated in Fig. 1) in order to witness the message $\langle j, snj\rangle$. This READY message will contribute to the time barrier needed to order $\langle j, snj\rangle$ with respect to other concurrent bscd-broadcast messages (more on this below). p_i then invokes try_deliver() (line 6) to see if it can bscd-deliver a set of messages.

When a process p_i bfifo-delivers a witness message READY($\langle j, snj\rangle$) from p_k regarding a message $\langle j, snj\rangle$, p_i uses the sequence number of the READY message (called snk, line 7) to update $witness_i[j, snj]$, the time barrier of the message $\langle j, snj\rangle$. Concretely, if $witness_i[j, snj][k] = +\infty$, p_i learns that p_k has witnessed the message identified by $\langle j, snj\rangle$ from p_j, and p_i records the time point at which this happened by updating $witness_i[j, snj][k]$ to snk (line 8). Then, if p_i has not yet bfifo-delivered the message $\langle j, snj\rangle$ (first predicate of line 9) and at least $t + 1$ processes (i.e., at least one correct process) have witnessed this message (by bfifo-broadcasting a READY($\langle j, snj\rangle$) message, second predicate of line 9), it knows that p_j bscd-broadcast a message identified by $\langle j, snj\rangle$ and it adds snj to $pending_i[j]$ (line 10). This early update of $pending_i[j]$ (i.e. before p_i receives the corresponding INIT message from p_j) ensures $\langle j, snj\rangle$ will be taken into account in the tests performed by try_deliver () at lines 12-16 to decide which application messages can be safely bscd-delivered (we return to these tests just below). Finally, after these various updates, p_i invokes try_deliver() (line 11) to attempt to bscd-deliver a set of messages.

Internal operation try_deliver () and predicate safe ()

In this operation, p_i first computes the set $candidates_i$ that contains the message identities $\langle j, snj\rangle$ that have been received but not yet delivered ($snj \in pending[j]$), and that have been witnessed by at least one correct process (first and second predicate of line 12). Then, to obtain the reduced set $todel_i$ (line 13) of messages that can actually be delivered, p_i first purges from $candidates_i$ the message identities whose payload has not been received yet or that have been witnessed by less than $(n-t)$ processes (a process p_k witnesses an application message m by bfifo-broadcasting READY($\langle j, snj\rangle$) at line 5 or 23, whose reception entails the update of $witness_i[j, snj]$ by p_i at line 8, see above).

Once this first purge is complete, p_i then removes from $todel_i$ (line 15) all the message identities $\langle j, snj\rangle$ (line 14) that make it "unsafe" with respect to messages in $candidates_i \setminus todel_i$ (the messages that have been "confirmed", but cannot be delivered yet, line 14), where *unsafe* is defined as the negation of the predicate safe() defined at lines 26-28. Thus a process will only bscd-deliver a set of messages $todel_i$ whose messages are all *safe* to bscd-deliver before those remaining in $candidates_i \setminus todel_i$. More precisely a message m identified by $\langle j, snj\rangle$ is considered *safe* by p_i to bscd-deliver before a message m' identified by $\langle j', snj'\rangle$ if (i) both have been bscd-broadcast by the same process with $snj < snj'$ (line 26) or (ii) p_i knows that a majority of processes witnessed m before m' (line 27). Thus a process may *learn* that it is safe to deliver m before m once it has received enough READY messages for m before READY messages from m' by the same senders. Note that this *safety* relationship is not transitive, and can lead to complex entanglements. For instance, Figure 2 shows the time barriers of three messages, i.e. the times at which correct processes and possibly a subset of Byzantine processes have first bfifo-broadcast READY($m_{\{A,B,C\}}$) messages. In this example, a process may learn that m_A is safe to bscd-deliver before m_B, and that m_B is safe to bscd-deliver before m_C, but never that m_A is safe to bscd-deliver before m_C. Thus correct processes must bscd-deliver m_A, m_B and m_C in the same set.

Finally, if $todel_i$ is not empty (line 17), p_i computes from this set of message identities the triplets $\langle j, snj, data_i[j, snj]\rangle$ (line 18), which define the set of messages ms that it can bscd-deliver (line 19). Then, according to the messages it has just bscd-delivered, p_i updates the sets $pending_i[j]$ (lines 20-21). Moreover, if such a set remains non-empty, and p_i bfifo-delivered the protocol message INIT(m) associated with min($pending_i[j]$) (line 22), p_i bfifo-broadcasts the protocol message READY(j, min($pending_i[j]$) to inform the other processes that this is the identity of the next message it intends to bscd-deliver from p_j (line 24).

▶ **Theorem 1.** *Algorithm 2 respects the properties of* BSCD-Validity*,* BSCD-Integrity*,* BSCD-Ordering*,* BSCD-Termination-1*, and* BSCD-Termination-2*.*

The following section is devoted to proving this result.

3.4 Proof of Correctness

We start by several intermediate definitions and results before the final proof of Theorem 1. The most important results are shown in Theorem 18 (BSCD-Order) and Theorem 22 (BSCD-Termination).

▶ **Definition 2.** *We say a correct process p_i bscd-delivers a message m from p_j with sequence number snj if it bscd-delivers a message set ms that contains the tuple $\langle j, snj, m\rangle$.*

▶ **Theorem 3.** *If a correct process p_i bscd-delivers a message m from a correct process p_j, then p_j bscd-broadcast m.*

Proof. If a correct process p_i bscd-delivers a set containing a message m from p_j, then m has been put in $data_i[j, snj]$ for some snj, meaning p_i bfifo-delivered $\text{INIT}(m)$ from p_j. By BFIFO-Validity, if p_j is correct then it bfifo-broadcast $\text{INIT}(m)$, meaning it bscd-broadcast m. ◀

▶ **Definition 4.** *We say a message identity $\langle j, snj\rangle$ is valid if all correct processes bfifo-deliver a message $\text{INIT}(m)$ from p_j at sequence number snj. Otherwise the message identity $\langle j, snj\rangle$ is invalid.*

▶ **Remark 5.** *If m is a message bscd-broadcast or bscd-delivered by a correct process, then we note $\text{id}(m) = \langle j, snj\rangle$ its identity which is by definition a valid identity. We will note $\text{sender}(m) = j$ and $\text{sn}(m) = snj$.*

▶ **Lemma 6.** *If at some point $snj \in pending_i[j]$ for a correct process p_i, then $\langle j, snj\rangle$ is a valid message identity, i.e. all correct processes will bfifo-deliver a message $\text{INIT}(m)$ from p_j with sequence number snj.*

Proof. If snj is added to $pending_i[j]$ at line 4 then p_i bfifo-delivered a message $\text{INIT}(m)$ from p_j at snj, so by BFIFO-Termination so will all other correct processes.

If snj is added to $pending_i[j]$ at line 10 then p_i bfifo-delivered $\text{READY}(\langle j, snj\rangle)$ from at least $t + 1$ processes, one of which at least is correct. Thus there is a correct process that bfifo-broadcast $\text{READY}(\langle j, snj\rangle)$, meaning it bfifo-delivered $\text{INIT}(m)$ from p_j at snj, so by BFIFO-Termination, all other correct processes also will. ◀

▶ **Theorem 7.** *No correct process bscd-delivers several (identical or different) messages for a given message identity $\langle j, snj\rangle$.*

Proof. A message that is bscd-delivered by a correct process p_i has a unique identity $\langle j, snj\rangle$ guaranteed by the bfifo-broadcast of the corresponding $\text{INIT}(m)$ message at p_j. At a correct process p_i, BFIFO-Integrity guarantees that $\text{INIT}(m)$ will be delivered only once with sequence number snj from p_j. snj will be added in $pending_i[j]$ at that time or before but cannot be added again later, and will be removed from $pending_i[j]$ only after $\text{INIT}(m)$ has been bfifo-delivered, therefore snj can only be removed from $pending_i[j]$ once meaning that m can be bscd-delivered only once. ◀

▶ **Lemma 8.** *If at any correct processes p_i at any time of its execution, and for any p_j, snj, p_k we have $witness_i[j, snj][k] = snk \neq \infty$ then snk is the sequence number of the first $\text{READY}(\langle j, snj\rangle)$ message bf-delivered by p_i from p_k, which is the same at every correct process, even if p_k is Byzantine.*

Proof. If $witness_i[j, snj][k]$ is set to other than ∞ it is when p_i bfifo-delivers $\text{READY}(\langle j, snj\rangle)$ from p_k. If p_i bfifo-delivers $\text{READY}(\langle j, snj\rangle)$ from p_k several times, the first time will lead to the execution of $witness_i[j, snj][k] \leftarrow snk$ and the second one won't because we now have $witness_i[j, snj][k] \neq \infty$ (line 8). Correct processes all bfifo-deliver the same messages from p_k in the same order and with the same sequence numbers, therefore they will eventually all update their $witness_i[j, snj][k]$ variable to the same value. ◀

▶ **Convention 9.** *Since only the first $\text{READY}(\langle j, snj\rangle)$ bfifo-delivered by a correct process p_i from p_k can lead to a change in $witness_i[j, p_j][k]$, we will say that p_i "received" $\text{READY}(\langle j, snj\rangle)$ from p_k to refer only to the first such READY bfifo-delivered. This is important if the sender of the READY, p_k, is Byzantine since in that case it may bfifo-broadcast several times the same READY message. We will say that p_i received $\text{READY}(\langle j, snj\rangle)$ before $\text{READY}(\langle j', snj'\rangle)$ from p_k*

if p_i bfifo-delivered for the first time READY$(\langle j, snj\rangle)$ *from p_k before bfifo-delivering for the first time* READY$(\langle j', snj'\rangle)$ *from p_k. We will also say that a correct process receives* READY$(\langle j, snj\rangle)$ *before* READY$(\langle j', snj'\rangle)$ *from p_k if it bfifo-delivers at least once* READY$(\langle j, snj\rangle)$ *from p_k and never bfifo-delivers* READY$(\langle j', snj'\rangle)$ *from p_k.*

▶ **Remark 10.** *If a correct process receives* READY$(\langle j, snj\rangle)$ *before* READY$(\langle j', snj'\rangle)$ *from a process p_k, then the same happens for all other correct processes: if p_i only bfifo-delivers one or more* READY$(\langle j, snj\rangle)$ *from p_k and no* READY$(\langle j', snj'\rangle)$ *from p_k then all other correct processes will also bfifo-deliver it or them and they will not bfifo-deliver a* READY$(\langle j', snj'\rangle)$ *from p_k. If p_i both bfifo-delivers one or more* READY$(\langle j, snj\rangle)$ *from p_k and one or more* READY$(\langle j', snj'\rangle)$ *from p_k, other correct processes will also bfifo-deliver them, in the same order and with the same sequence numbers.*

▶ **Lemma 11.** *If at any correct process p_i, for any p_j, snj, p'_j, snj' and p_k, we have $witness_i[j, snj][k] < witness_i[j', snj'][k]$ at any time of its execution, then that will always be the case at p_i afterwards.*

Proof. If $witness_i[j, snj][k] < witness_i[j', snj'][k]$ then $witness_i[j, snj][k]$ was set to a non-infinite value which is the sequence number with which p_i bfifo-delivers the first READY$(\langle j, snj\rangle)$ from p_k. The content of $witness_i[j, snj][k]$ will never be changed afterwards, and if $witness_i[j', snj'][k]$ is changed afterwards it will be set to the sequence number with which p_i bfifo-delivers the first READY$(\langle j', snj'\rangle)$ from p_k which is necessarily larger than the sequence number of the first READY$(\langle j, snj\rangle)$. ◀

We now contruct a relation $\rightarrow$ on messages to capture the fact that a message m can be safely bscd-delivered before a message m': if $m \rightarrow m'$, no correct process delivers m' before m. The relation $\rightarrow$ relies on the time barriers of m and m' arising from the witness messages READY. We construct $\rightarrow$ incrementally, starting with a simpler preliminary relation $\prec_k$.

▶ **Definition 12.** *If correct processes receive* READY$(\langle j, snj\rangle)$ *before* READY$(\langle j', snj'\rangle)$ *from p_k, we write: $\langle j, snj\rangle \prec_k \langle j', snj'\rangle$.*

▶ **Remark 13.** *If p_i is a correct process, $witness_i[j, snj][k] < witness_i[j', snj'][k]$ can be interpreted as: "p_i knows that $\langle j, snj\rangle \prec_k \langle j', snj'\rangle$" and implies $\langle j, snj\rangle \prec_k \langle j', snj'\rangle$.*

▶ **Definition 14.** *Let us note $m \rightarrow m'$ if* (i) *there is a majority P of processes such that $\forall p_k \in P, m \prec_k m'$, or if* (ii) *$m$ and m' are sent by the same process and* $\mathrm{sn}(m) < \mathrm{sn}(m')$. *Formally:* $m \rightarrow m' \;\Leftrightarrow\; \Big(\mathrm{sender}(m) = \mathrm{sender}(m') \wedge \mathrm{sn}(m) < \mathrm{sn}(m')\Big) \vee \Big(|\{k : m \prec_k m'\}| > \frac{n}{2}\Big)$.

▶ **Lemma 15.** *$m \rightarrow m'$ and $m' \rightarrow m$ are exclusive: $\forall m, m', \neg(m \rightarrow m' \wedge m' \rightarrow m)$.*

Proof. If $\mathrm{sender}(m) = \mathrm{sender}(m')$, then suppose w.l.o.g. that $\mathrm{sn}(m) < \mathrm{sn}(m')$. We directly have $m \rightarrow m'$ following from the definition. Moreover, if correct processes bfifo-broadcast READY$(\mathrm{id}(m'))$, they will do it after they have already bfifo-broadcast READY$(\mathrm{id}(m))$, thus $|\{k : m' \prec_k m\}| \leq t$, thus we have $\neg(m' \rightarrow m)$.

If m and m' are not from the same sender and $m \rightarrow m'$ and $m \rightarrow m'$ then we have $|\{k : m \prec_k m'\}| > \frac{n}{2}$ and $|\{k : m' \prec_k m\}| > \frac{n}{2}$. There is a process in the intersection of these two majorities for which we have a contradiction. ◀

▶ **Remark 16.** *$\rightarrow$ is not a partial order as it is not transitive.*

▶ **Lemma 17.** *If a correct process p_i bscd-delivers m before bscd-delivering m' or bscd-delivers m and bfifo-delivers* INIT(m') *but never bscd-delivers m' then $m \rightarrow m'$.*

Proof. When m is bscd-delivered by p_i, it has received $\text{READY}(\text{id}(m))$ from at least $n-t$ processes. Two cases arise:

- Case 1: If p_i has already received an $\text{READY}(\text{id}(m'))$ from no more than t processes, then there are at least $n-2t$ processes for which p_i has received a READY $(\text{id}(m))$ and not yet a $\text{READY}(\text{id}(m'))$, therefore $|\{k : m \prec_k m'\}| \geqslant n-2t$. The hypothesis of our computing model is $t < n/4$, therefore $n-2t > \frac{n}{2}$. We thus have $m \to m'$.
- Case 2: If p_i has already received an $\text{READY}(\text{id}(m'))$ from $t+1$ or more processes when it bscd-delivered m, then $\text{sn}(m') \in \mathit{pending}_i[k]$ (because of line 9), thus $\text{id}(m') \in \mathit{candidates}_i \backslash \mathit{todel}_i$, thus by the exit condition of the while loop, either $\text{sender}(m) = \text{sender}(m') \wedge \text{sn}(m) < \text{sn}(m')$ or $|\{k : \mathit{witness}_i[\text{id}(m)][k] < \mathit{witness}_i[\text{id}(m')][k]\}| > \frac{n}{2}$, in both cases $m \to m'$. ◀

▶ **Theorem 18.** *Let p_i be a correct process that bscd-delivers a set ms_i containing a message m and later bscd-delivers a set ms'_i containing a message m'. No correct process p_j bscd-delivers first a set ms'_j containing m' and later a message set ms_j containing m.*

Proof. By Lemma 17, such a situation would imply $m \to m'$ and $m' \to m$, which we have shown are exclusive (Lemma 15). ◀

▶ **Definition 19.** *We say that a correct process p_i knows that $m \to m'$ if either $|\{k : \mathit{witness}_i[\text{id}(m)][k] < \mathit{witness}_i[\text{id}(m')][k]\}| > \frac{n}{2}$ or $\text{sender}(m) = \text{sender}(m') \wedge \text{sn}(m) < \text{sn}(m')$.*

▶ **Remark 20.** *If a correct process p_i knows that $m \to m'$ then $m \to m'$.*

▶ **Remark 21.** *In the while loop of* try_deliver(), *a correct process removes a message m from todel_i when there is a message m' in $\mathit{candidates}_i \backslash \mathit{todel}_i$ for which it does not know that $m \to m'$. For a correct process to bscd-deliver a set of messages ms, it must know that $m \to m'$ for all $m \in \text{ms}$ and $m' \in \mathit{candidates}_i \backslash \text{ms}$.*

▶ **Theorem 22.** *If a correct process p_i bfifo-delivers $\text{INIT}(m)$ from p_j, it eventually bscd-delivers m from p_j.*

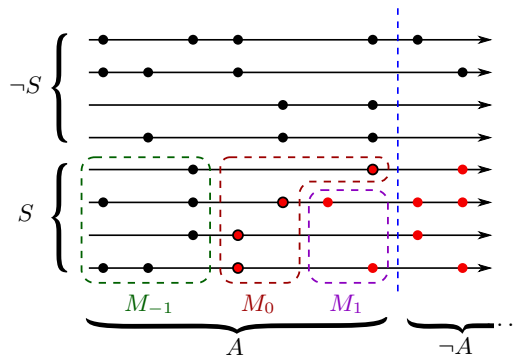


Figure 3 Message sets used in the proof of Theorem 22.

Proof. Suppose by contradiction that there is a correct process that bfifo-delivers an $\text{INIT}(m)$ but never bscd-delivers m. Let M be the set of these *unterminated* messages, i.e. the set of messages m for which correct processes bfifo-deliver $\text{INIT}(m)$ but there is at least one correct process that does not bscd-deliver m.

Let $M_{p_i} = \{m \in M : \text{sender}(m) = p_i\}$, the set of unterminated messages bscd-broadcast by p_i, and $S = \{i : M_{p_i} \neq \varnothing\}$, the set of processes who bscd-broadcast at least one unterminated message. For $i \in S$, let m_i be the message in M_{p_i} with the lowest sequence number, and let $M_0 = \{m_i, i \in S\}$. We use M_0 as a boundary to define four sets of messages that will lead us to a contradiction. Let $M_{-1} = \{m : \text{sender}(m) \in S \wedge \text{sn}(m) < \text{sn}(m_i)\}$ the set of all messages by processes of S that all correct processes bscd-deliver.

After some time, correct processes all bscd-deliver the messages of M_{-1}, thus all correct processes will bfifo-broadcast $\textsc{ready}(\text{id}(m_i))$ for all $i \in S$ after some time. Let A be the set of messages witnessed (with a READY message) by at least one correct process before one of the unterminated messages in the "boundary" M_0. More formally A is defined by: $m' \in A$ iff a correct process bfifo-broadcasts $\textsc{ready}(\text{id}(m'))$ before bfifo-broadcasting $\textsc{ready}(\text{id}(m_i))$ for some $i \in S$. Because a process may only receive a finite number of messages in a finite amount of time, the set A is finite. For all $m \notin A$ and $i \in S$ we have $m_i \to m$.

Let $A_S = \{m \in A : \text{sender}(m) \in S\}$ and $A_{\neg S} = \{m \in A : \text{sender}(m) \notin S\}$. Let $M_1 = A_S \backslash (M_{-1} \cup M_0)$. We have that $A = A_{\neg S} \cup M_{-1} \cup M_0 \cup M_1$. We have two cases:

- Case 1: no correct process ever bscd-delivers a message of M_0. In that case, no correct process will ever bfifo-broadcast a READY for a message of M_1, thus the messages of M_1 will never be considered in the $\mathit{candidates}_i$ set of correct processes.

  Let p_i be a correct process. After a certain time p_i will have bscd-delivered all messages of $A_{\neg S}$ and of M_{-1} and will have bfifo-delivered its last $\textsc{ready}(\text{id}(m))$ for a message $m \in M_0$, thus it will execute $\mathsf{try_deliver}()$ either strictly after all that happens or when bscd-delivering for the last time messages of $A_{\neg S} \cup M_{-1}$ and after having bfifo-delivered its last $\textsc{ready}(\text{id}(m))$ for a message $m \in M_0$. At that point $\mathit{candidates}_i$ will contain only messages of M_0 and messages not in A, and possibly some messages of $A_{\neg S} \cup M_{-1}$ which will be in todel_i at the end of the while loop. Since p_i never bscd-delivers messages of M_0, at some point in the while loop it will remove a message $m \in M_0$.

  Let m be the first message of M_0 p_i removes from todel_i, it is removed because there is an $m' \in \mathit{candidates}_i \backslash M_0$ that violates the exit condition of the loop. We have $m' \notin M_0$ because m is the first message of M_0 that p_i removes from todel_i and initially there is no message of M_0 in $\mathit{candidates}_i \backslash \mathit{todel}_i$, $m' \notin A_{\neg S} \cup M_{-1}$ because all messages of $A_{\neg S} \cup M_{-1}$ remaining at this step are in todel_i at the end of the loop, and $m' \notin M_1$ because no message of M_1 is in $\mathit{candidates}_i$, thus $m' \notin A$. Since $m \in M_0$ and $m' \notin A$ we have $m \to m'$ and since p_i has bfifo-delivered all messages $\textsc{ready}(\text{id}(m))$ then it knows that $m \to m'$, meaning that $\mathsf{safe}(\text{id}(m), \text{id}(m')) = \texttt{true}$. Therefore p_i cannot remove m from todel_i because of m'. We have a contradiction.

- Case 2: there is a correct process p_i that bscd-delivers a message $m \in M_0$. In that case, p_i delivered a certain message set ms that contains m. Let us note U the set of all messages p_i bscd-delivered before and including ms. For any message m' for which correct processes deliver $\textsc{init}(m')$ and such that $m' \notin U$, by Lemma 17 we have $\forall m'' \in U, m'' \to m'$.

  Let p_j be a correct process that does not bscd-deliver m. After a certain time, it will receive its last READY for a message of U (in particular, it will have bfifo-delivered $\textsc{ready}(\text{id}(m))$ from all correct processes), at which time it executes $\mathsf{try_deliver}()$ and removes at least one message of U from todel_j (it removes at least m). Moreover no messages of U are in $\mathit{candidates}_i \backslash \mathit{todel}_i$ at the beginning of the loop since p_i was able to bscd-deliver all messages of U and all READY messages for messages of U have been bfifo-delivered by p_j. Let m_0 be the first message of U that p_j removes from todel_j. The message m_0 is removed because of some message $m_1 \notin U$ that violated the condition. Since p_i bscd-delivered m_0 before bscd-delivering m_1 we have $m_0 \to m_1$. Since p_j has

already bfifo-delivered its last messages READY(id(m_0)) then it knows that $m_0 \rightarrow m_1$, meaning that safe(id(m), id(m')) = true. Therefore p_j cannot remove m from $todel_i$ because of m'. We have a contradiction. ◀

Final proof of Theorem 1

Proof. BSCD-Validity, BSCD-Integrity and BSCD-Ordering are shown respectively in Theorem 3, Theorem 7 and Theorem 18. BSCD-Termination-1: if a correct procsess bscd-broadcasts a message m, then it will bf-broadcast INIT(m). By BFIFO-Termination it will thus bf-deliver INIT(m), thus by Theorem 22 it will bscd-deliver m. BSCD-Termination-2: If a correct process bscd-delivers m then it has previously bf-delivered INIT(m). By BFIFO-Termination all correct processes bf-deliver INIT(m). By Theorem 22, all correct processes will bscd-deliver m. ◀

4 BSCD-broadcast in action: a Byzantine-tolerant snapshot object

The snapshot object was introduced in [1, 2]. A snapshot object can be seen as an array $REG[1..n]$ of single-writer/multi-reader atomic registers which provides processes with two operations, denoted write() and snapshot(). The invocation of write(v) by a process p_i assigns atomically v to $REG[i]$. The invocation of snapshot() returns the value of $REG[1..n]$ as if it was executed instantaneously. Hence, in any execution of a snapshot object, its operations write() and snapshot() are linearizable [7]. As mentioned in the introduction, the snapshot object has recently been shown to be instrumental in the design of cryptocurrency systems [5].

Byzantine Snapshot object on top of BSCD-broadcast

Let us recall that nothing can prevent Byzantine processes from writing fake values in their register of the snapshot object. The important property is that the registers associated with correct processes cannot be corrupted by Byzantine processes.

Our algorithm is similar in structure to the multi-writer/multi-reader snapshot object built on SCD-broadcast that was introduced in [8] in the context of process crash failures. However a multi-writer snapshot object is intrinsically unsuited to a system with Byzantine processes as such processes would then be able to prevent all effective communication between correct ones by overwriting correct processes' values as soon as they are written, which is why we focus on a single-writer/multi-reader snapshot object instead. As a consequence, the analysis of our algorithm shares little in common with that of [8].

Local data structures at a process p_i

Let REG be the snapshot object. At a process p_i, it uses three local data structures:

- $done_i$ is a Boolean variable.
- $reg_i[1..n]$ is the value of $REG[1..n]$ as currently known by p_i.
- $wsn_i[1..n]$ is such that $wsn_i[j]$ is the sequence number of the last write by p_j in $REG[j]$, as known by p_i. Such a sequence number is systematically associated (at line 18 of Algorithm 2) with each invocation of bscd_broadcast WRITE() (line 3 of Algorithm 3).

Operation snapshot()

When a process invokes snapshot(), it simply invokes bscd_broadcast SYNC(), where SYNC is a synchronization tag, and (with the help of the Boolean $done_i$, lines 1 and 7) waits until this message has been locally processed. Then, p_i returns the value of the local array reg_i.

Algorithm 3 Construction of a snapshot object on top of BSCD-broadcast (code for p_i).

init $reg_i \leftarrow [\perp, \ldots, \perp]$; $wsn_i \leftarrow [0, \ldots, 0]$.

operation snapshot() **is**
(1) $done_i \leftarrow$ `false`; bscd_broadcast SYNC(); wait($done_i$);
(2) return($reg_i[1..n]$).

operation write(v) **is**
(3) $done_i \leftarrow$ `false`; bscd_broadcast WRITE(v); wait($done_i$).

when the message set $\{ \langle j_1, sn_1, \text{WRITE}(v_1)\rangle, \ldots, \langle j_x, sn_x, \text{WRITE}(v_x)\rangle, \langle j_{x+1}, sn_{x+1}, \text{SYNC}()\rangle, \ldots, \langle j_y, sn_y, \text{SYNC}()\rangle \}$ **is bscd-delivered do**
(4) **for each message** $\langle j, snj, \text{WRITE}(v)\rangle \in$ bscd-delivered message set **do**
(5) **if** $(wsn_i[j] < snj)$ **then** $reg_i[j] \leftarrow v$; $wsn_i[j] \leftarrow snj$ **end if**
(6) **end for**;
(7) **if** $\exists \ell : j_\ell = i$ **then** $done_i \leftarrow$ `true` **end if**.

The aim of the message SYNC is to stop the progress of p_i so that, once unblocked, it will have a consistent value of REG it can return ("consistent" refers here to the atomicity of the snapshot object).

Operation write()

The code of this operation is similar in structure to the synchronization of operation snapshot(), however instead of reading the registers and returning their values, the process sends a value to be written but does not return anything. When a process invokes write(v), it simply invokes bscd_broadcast WRITE(v), where WRITE is an operation tag, and, with the help of the Boolean $done_i$, waits until this message has been locally processed (lines 3 and 7, this synchronization pattern is sometimes called "read your writes").

Processing of a set of messages

This procedure consists in two steps:

- Process p_i first considers the messages WRITE() that appear in the message set ms it is bscd-delivering. Each element of ms is actually a triplet $\langle j, snj, v\rangle$, such that $\langle j, snj\rangle$ is the identity of the value v, namely the snj-th value written by p_j in $REG[j]$. p_i writes v into $reg_i[j]$, if this value has not been overwritten by a more recent value (lines 4-6).
- After the previous updates, p_i sets the Boolean $done_i$ to the value `true` if the message set contains a message it bscd-broadcast (line 7).

▶ **Theorem 23.** *Algorithm* 3 *builds a linearizable* snapshot *object in an n-process asynchronous message-passing system where up to $t < n/4$ processes may commit Byzantine failures.*

The proof of this theorem can be found in Appendix B.2.

5 Conclusion

This paper addressed the design of a Set-Constrained Delivery broadcast abstraction in the context of n-process asynchronous message-passing systems where up to t processes may commit Byzantine failures. A first primitive, BFIFO-broadcast, ensures that, for any sender, all correct processes deliver its messages in the same order (which is their sending order if the sender is correct). BSCD-broadcast, which is built over BFIFO-broadcast, ensures that correct processes deliver sets of messages such that, if a correct process p delivers a

set of messages containing a message m and later delivers a set of messages containing a message m', no correct process delivers first a set of messages containing m' and later a set of messages containing m. As an illustration of BSCD-broadcast, it has been shown how it facilitates the construction of a Byzantine-tolerant read/write snapshot object.

References

1. Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic Snapshots of Shared Memory. *J. ACM*, 40(4):873–890, 1993. doi:10.1145/153724.153741.
2. James H. Anderson. Multi-Writer Composite Registers. *Distributed Computing*, 7(4):175–195, 1994. doi:10.1007/BF02280833.
3. Gabriel Bracha. Asynchronous Byzantine Agreement Protocols. *Inf. Comput.*, 75(2):130–143, 1987. doi:10.1016/0890-5401(87)90054-X.
4. Thomas D. Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. Adding Concurrency to Smart Contracts. In Elad Michael Schiller and Alexander A. Schwarzmann, editors, *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, pages 303–312. ACM, 2017. doi:10.1145/3087801.3087835.
5. Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. The Consensus Number of a Cryptocurrency. In Peter Robinson and Faith Ellen, editors, *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019.*, pages 307–316. ACM, 2019. doi:10.1145/3293611.3331589.
6. Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Cornell University, 1994.
7. Maurice Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. doi:10.1145/78969.78972.
8. Damien Imbs, Achour Mostéfaoui, Matthieu Perrin, and Michel Raynal. Set-Constrained Delivery Broadcast: Definition, Abstraction Power, and Computability Limits. In Paolo Bellavista and Vijay K. Garg, editors, *Proceedings of the 19th International Conference on Distributed Computing and Networking, ICDCN 2018, Varanasi, India, January 4-7, 2018*, pages 7:1–7:10. ACM, 2018. doi:10.1145/3154273.3154296.
9. Damien Imbs and Michel Raynal. Trading off t-Resilience for Efficiency in Asynchronous Byzantine Reliable Broadcast. *Parallel Processing Letters*, 26(4):1–8, 2016. doi:10.1142/S0129626416500171.
10. Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982. doi:10.1145/357172.357176.
11. Satoshi Nakamoto et al. Bitcoin: A peer-to-peer electronic cash system, 2008.
12. Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching Agreement in the Presence of Faults. *J. ACM*, 27(2):228–234, 1980. doi:10.1145/322186.322188.
13. Michel Raynal. *Fault-Tolerant Message-Passing Distributed Systems - An Algorithmic Approach*. Springer, 2018. doi:10.1007/978-3-319-94141-7.
14. Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-Free Replicated Data Types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400. Springer, 2011. doi:10.1007/978-3-642-24550-3_29.

A Two Multi-shot Signature-free BR-broadcast Algorithms

In order to make the paper as self-contained as possible, this section presents multi-shot extensions of the one-shot signature-free BR-broadcast algorithms introduced by G. Bracha [3] and D. Imbs and M. Raynal [9]. The presentation follows pages 64-71 of [13], where the reader can also find proofs of these algorithms. In the text of these two extensions, sn denotes the sequence number of the corresponding BR-broadcast instance, hence a process invokes br_broadcast(sn, m).

A.1 Underlying Basic Communication System

In both algorithms described below, the processes communicate by exchanging messages through an asynchronous reliable point-to-point network. "Asynchronous" means that a message that has been sent is eventually received by its destination process, i.e., there is no bound on message transfer delays. "Reliable" means that the network does not loose, duplicate, modify, or create messages. "Point-to-point" means that there is a bi-directional communication channel between each pair of processes. As a consequence, a process can identify the sender of each message it receives and no Byzantine process can impersonate another process. In practice, this means that Byzantine processes cannot control the underlying communication layer.

A process p_i sends a message to a process p_j by invoking the primitive "send TAG(m) to p_j", 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. A process receives a message by executing the primitive "receive()". The macro-operation "broadcast TAG(m)" is a shortcut for "**for** $j \in \{1, \cdots, n\}$ **do** send TAG(m) to p_j **end for**".

A.2 Multi-shot Version of Bracha's BR-broadcast Algorithm

Algorithm 4 is a multi-shot version of Bracha's BR-broadcast algorithm. This algorithm assumes $t < n/3$. When, on its client side, a process p_i invokes br_broadcast(sn, m), it invoke the macro-operation broadcast() with the protocol message INIT(sn, m) (line 1).

Algorithm 4 Multi-shot version of Bracha's BR-broadcast algorithm ($t < n/3$, code for p_i).

operation br_broadcast(sn, m) **is**
(1) broadcast INIT(sn, m).

when a message INIT(sn, m) **is received from** p_j **do**
(2) discard the message if it is not the first message INIT$(sn, -)$ received from p_j;
(3) broadcast ECHO$(\langle j, sn\rangle, m)$.

when a message ECHO$(\langle j, sn\rangle, m)$ **is received from any process do**
(4) **if** (ECHO$(\langle j, sn\rangle, m)$ received from strictly more than $\frac{n+t}{2}$ different processes)
$\wedge$ (READY$(\langle j, sn\rangle, m)$ not yet broadcast)
(5) **then** broadcast READY$(\langle j, sn\rangle, m)$
(6) **end if**.

when a message READY$(\langle j, sn\rangle, m)$ **is received from any process do**
(7) **if** (READY$(\langle j, sn\rangle, m)$ received from at least $(t + 1)$ different processes)
$\wedge$ (READY$(\langle j, sn\rangle, m)$ not yet broadcast)
(8) **then** broadcast READY$(\langle j, sn\rangle, m)$
(9) **end if**;
(10) **if** (READY$(\langle j, sn\rangle, m)$ received from at least $(2t + 1)$ different processes)
$\wedge$ ($\langle j, sn, m\rangle$ not yet br_delivered)
(11) **then** br_deliver $\langle j, sn, m\rangle$
(12) **end if**.

On it server side a process p_i may receive three different types of protocol messages: INIT(), ECHO(), and READY(). A message INIT carries an application message, while the messages ECHO() and READY() carry a process identity and an application message[1].

- When p_i receives INIT(sn, m) for the first time from a process p_j (line 2), it broadcasts the protocol message ECHO$(\langle j, sn\rangle, m)$ (line 3). If this message is not the first message INIT$(sn, -)$ from p_j, p_i discards it (in this case, p_j is Byzantine).
- When p_i receives the protocol ECHO$(\langle j, sn\rangle, m)$ from any process, it broadcasts the protocol message READY$(\langle j, sn\rangle, m)$ (line 5) if it received ECHO$(\langle j, sn\rangle, m)$ from enough different processes (where "enough" means here more than $\frac{n+t}{2}$), and READY$(\langle j, sn\rangle, m)$ has not yet been broadcast (line 4). This message exchanges ensure that no two correct processes will br-deliver different message from p_j with the sequence number sn, but it is still possible that a correct process br-delivers m from p_j while another correct process does not br-deliver a message fromp_j. The role of the message READY$(\langle j, sn\rangle, m)$ is to prevent a correct process from blocking on the br-delivery of m.
- When p_i receives READY$(\langle j, sn\rangle, m)$ for any process, it does the following.
  - Process p_i first broadcasts READY$(\langle j, sn\rangle, m)$ (line 8) if (i) not already done and (ii) it received READY$(\langle j, sn\rangle, m)$ for "enough" processes (where "enough" means here $(t + 1)$ processes, which means from at least on correct process, line 7). As previously indicated, this allows other correct processes not to deadlock.
  - Then, if p_i received READY$(\langle j, sn\rangle, m)$ from "enough" processes (where "enough" means here $((2t+1)$, which means from at least $(t+1)$ correct processes), it locally br-delivers the pair (sn, m) (from p_j), if not yet already done (lines 10-11).

This algorithm is optimal with respect to t-resilience (namely $t < n/3$). It requires three consecutive communication steps, and $(n - 1) + 2n(n - 1) = 2n^2 - n - 1$ protocol messages. The proof of this algorithm relies on the following properties, which assume $n > 3t$ (see [13] for their proofs):

- $n - t > \frac{n+t}{2}$.
- Any set containing more than $\frac{n+t}{2}$ different processes, contains at least $(t + 1)$ non-faulty processes.
- Any two sets of processes Q_1 and Q_2 of size at least $\lfloor \frac{n+t}{2} \rfloor + 1$ have at least one correct process in their intersection.

A.3 Multi-shot Version of Imbs-Raynal's BR-broadcast Algorithm

Algorithm 5 is a multi-shot version of Imbs-Raynal's BR-broadcast algorithm. This algorithm assumes $t < n/5$. The code of br_broadcast(sn, m) is the same as in the previous algorithm.

On its server side a process p_i may receive two different types of protocol messages: INIT() and WITNESS(). The processing of INIT(sn, m) is similar to the one of Algorithm 4. Process p_i simply broadcasts the message WITNESS$(\langle j, sn\rangle, m)$ if it is the first time it received from p_j a message INIT$(sn, -)$ (line 3). Then, when it receives a message WITNESS$(\langle j, sn\rangle, m)$ p_i does the following.

- If it received the same message WITNESS$(\langle j, sn\rangle, m)$ from "enough" processes (where "enough" means here $n - 2t$), and it has not yet broadcast this message (line 4), it does it (line 5).

[1] The fact that the ECHO() and READY() messages carry a process identity makes redundant the use of an identity in the pair $\langle -, sn\rangle$ that appear in the messages that are br-broadcast (see the paragraph "Invocation pattern" at the end of Section 2.3).

Algorithm 5 Multi-shot version of Imbs-Raynal's BR-broadcast algorithm ($t < n/5$, code for p_i).

operation br_broadcast(sn, m) **is**
(1) broadcast INIT(sn, m).

when INIT(sn, m) **is received from** p_j **do**
(2) discard the message if it is not the first message INIT$(sn, -)$ received from p_j;
(3) broadcast WITNESS$(\langle i, sn\rangle, m)$.

when WITNESS$(\langle j, sn\rangle, m)$ **is received from any process do**
(4) **if** (WITNESS$(\langle j, sn\rangle, m)$ received from at least $(n - 2t)$ different processes)
$\wedge$ (WITNESS$(\langle i, sn\rangle, m)$ not yet broadcast)
(5) **then** broadcast WITNESS$(\langle j, sn\rangle, m)$
(6) **end if**;
(7) **if** (WITNESS$(\langle j, sn\rangle, m)$ received from at least $(n - t)$ different processes)
$\wedge$ ($\langle j, sn, m\rangle$ not yet br_delivered)
(8) **then** br_deliver $\langle j, sn, m\rangle$
(9) **end if**.

- If it received WITNESS$(\langle j, sn\rangle, m)$ from "more" processes (where "more" means here $n-t$), and it has not yet br-delivered the pair (sn, m) from p_j (line 7), it br-delivers it ((line 8).

Let us notice that, as $t < n/5$, we have $n - 2t > 3t$, which means that, in this case, (WITNESS(j, m) was broadcast by at least $n - 3t \geq 2t + 1$ correct processes. Then, if it received WITNESS(j, m) from more different processes, where "more" means" $(n-t)$, p_i locally br-delivers m from p_j.

As we can easily see, this algorithm requires two communication steps and $n^2 - 1$ protocol messages. This better efficiency with respect to Bracha's algorithm is obtained at the price of a weaker t-resilience, namely $t < n/5$.

B Proofs of Algorithms

This appendix contains the complete proofs for the theorems that were not proved in the paper.

B.1 BFIFO-broadcast on top of MBR-broadcast

BFIFO-Validity follows directly from MBR-Validity. BFIFO-Integrity (resp. BFIFO-Termination) follows directly from MBR-Integrity (resp. MBR-Termination), the delivery condition (line 3), and the increase of $fifo_del_i[j]$ (line 5). The formalization of these proofs are left to the reader. The next (easy) theorem concerns the BFIFO-Order property.

▶ **Theorem 24.** *Algorithm* 1 *satisfies the* BFIFO-Order *property.*

Proof. Suppose a correct process p_i bfifo-delivers first a message m, then a message m', both from the same sender p_j. These messages were br-delivered to p_i with some sequence numbers, sn and sn'. As p_i bfifo-delivers message from any process in the order defined by their sequence numbers, we have $sn < sn'$. It follows then from MBR-Termination-2 that all correct processes br-deliver m and m' with sequence numbers sn and sn', respectively. It follows that any correct process bfifo-delivers m before m'.

If the sender p_j of m and m' is correct, it associated the sequence number sn with m and the sequence number $sn' > sn$ with m'. It follows that p_j bfifo-broadcast m before m'. ◀

The computability and messages/time complexities of this algorithm are the ones of the underlying Byzantine reliable broadcast algorithm.

B.2 BSCD-broadcast in action: a Byzantine-tolerant snapshot object

We start by several intermediate definitions and results before the final proof of Theorem 23.

▶ **Lemma 25.** *If a correct process invokes an operation, it returns from its invocation.*

Proof. Let p_i be a correct process that invokes a read or write operation. By the BSCD-Termination-1 property of BSCD-broadcast, it eventually receives a message set containing the message $\langle i, sni, x\rangle$ which it sent at line 1 or 3 of Algorithm 3. As all the statements associated with the bscd-delivery of a message set terminate, it follows that the synchronization boolean $done_i$ is eventually set to `true`. Consequently, p_i returns from the invocation of its operation. ◀

▶ **Definition 26.** *A sequence number array* sna *is an array* $sna = [sna[1], \ldots, sna[n]]$ *of* n *sequence numbers, one per process.*

▶ **Remark 27.** *In Algorithm 3,* wsn_i *is a sequence number array that is used to keep track of* p_i*'s vision of the operations performed so far.*

▶ **Definition 28.** *Let* $\leq_{sna}$ *be the product order defined on sequence number arrays as:*

$$sna_1 \leq_{sna} sna_2 \iff \forall k \in 1..n,\ sna_1[k] \leq sna_2[k].$$

Let $<_{sna}$ *be the relation defined as:*

$$sna_1 <_{sna} sna_2 \iff (sna_1 \leq_{sna} sna_2) \wedge (sna_1 \neq sna_2).$$

▶ **Definition 29.** *If* p_i *is a correct process, let* WSN_i *be the set of the array values taken by* wsn_i *at line 7 during an execution, after the processing of message sets by process* p_i*. Let* $WSN = \bigcup_{p_i \text{ correct}} WSN_i$*.*

▶ **Lemma 30.** *The order* $<_{sna}$ *is total and well founded on* WSN*.*

Proof. Let us first observe that, for any correct process p_i, all values in WSN_i are totally ordered: this comes from wsn_i whose entries can only increase (line 5). Hence, let sna_1 be an array value of WSN_i and sna_2 be an array value of WSN_j with $i \neq j$ and where both p_i and p_j are correct processes.

Let us assume, by contradiction, that $\neg(sna_1 \leq_{sna} sna_2)$ and $\neg(sna_2 \leq_{sna} sna_1)$. Thus there is a k such that $sna_1[k] > sna_2[k]$ and a k' such that $sna_1[k'] < sna_2[k']$. According to lines 4 and 5, there is a message $\langle k, sna_1[k], \textsc{write}(v)\rangle$ that has been received by p_i when $wsn_i = sna_1$ and not by p_j when $wsn_j = sna_2$. Similarly, there is a message $\langle k', sna_2[k'], \textsc{write}(v')\rangle$ that has been received by p_j when $wsn_j = sna_2$ and not by p_i when $wsn_i = sna_1$. This situation directly contradicts the BSCD-Order property, from which we conclude that either $sna_1 \leq_{sna} sna_2$ or $sna_2 \leq_{sna} sna_1$. Therefore $<_{sna}$ is a total order.

Since all elements of WSN are vectors of elements of $\mathbb{N}$, they all have a finite number of strictly smaller elements, therefore $<_{sna}$ is well founded on WSN. ◀

▶ **Definition 31.** *Let* C *be the set of triples* $\langle i, sni, x\rangle$ *bscd-broadcast by correct processes when invoking an operation (line 1 or 3), where* i *and* sni *is the control information provided by the BSCD-broadcast abstraction and* x *is the value sent, either* $\textsc{sync}()$ *or* $\textsc{write}(v)$*.*

Since each operation invoked by a correct process corresponds to the BSCD-broadcast of a unique message, we will identify the corresponding message triple with the invoked operation.

For any $\langle i, sni, x\rangle, \langle j, snj, y\rangle \in C$*, we write* $\langle i, sni, x\rangle \prec \langle j, snj, y\rangle$ *if* p_i *returned from its operation that bscd-broadcast* $\langle i, sni, x\rangle$ *before* p_j *started its operation that bscd-broadcast* $\langle j, snj, y\rangle$*.*

▶ **Definition 32.** *Let W be the set of triples $\langle i, sni, \text{WRITE}(v)\rangle$ bscd-delivered by correct processes. This set may contain write operations by Byzantine processes.*

▶ **Definition 33.** *Let $O = W \cup C$ be the set of read (or SYNC) and write operations invoked by correct processes and of write operations received by correct processes from Byzantine processes.*

▶ **Remark 34.** *As a direct consequence of BSCD-broadcast properties, the set O contains at most one triplet of the form $\langle i, sni, -\rangle$ for given i and sni.*

▶ **Lemma 35.** $\langle i, sni, \text{WRITE}(v)\rangle \prec \langle i, sni', \text{WRITE}(v')\rangle \implies sni < sni'$.

Proof. $\prec$ only applies to operations by correct processes, therefore p_i is a correct process. If a correct process terminates WRITE(v) before starting WRITE(v'), then it bscd-broadcast WRITE(v) before WRITE(v'), therefore $sni < sni'$. ◀

▶ **Definition 36.** *If sna is a sequence number array, let $W(sna) = \{\langle i, sni, \text{WRITE}(v)\rangle \in W : sni \leqslant sna[i]\}$ be the set of write operations included in the time barrier defined by sna.*

▶ **Definition 37.** *If τ is a time at which a correct process p_i executes line 7, then:*
- *let $\text{wsn}_i(\tau)$ be the value of wsn_i when p_i executes line 7 at time τ*
- *let $U_i(\tau)$ be the union of all the message sets bscd-delivered by p_i until τ*

▶ **Lemma 38.** *$U_i(\tau) \cap W = W(\text{wsn}_i(\tau))$. As a consequence:*

$$U_i(\tau) \cap W \subseteq U_j(\tau') \cap W \iff \text{wsn}_i(\tau) \leqslant_{sna} \text{wsn}_j(\tau'),$$
$$U_i(\tau) \cap W \subsetneq U_j(\tau') \cap W \iff \text{wsn}_i(\tau) <_{sna} \text{wsn}_j(\tau').$$

Proof. Let $\langle j, snj, \text{WRITE}(v)\rangle$ be a message of W. If $\langle j, snj, \text{WRITE}(v)\rangle \in U_i(\tau)$, then when p_i executes line 7 at time τ, it has already bscd-delivered and processed $\langle j, snj, \text{WRITE}(v)\rangle$, therefore $\text{wsn}_i(\tau)[j] \geq snj$, therefore $\langle j, snj, \text{WRITE}(v)\rangle \in W(\text{wsn}_i(\tau))$. Conversely, if $\langle j, snj, \text{WRITE}(v)\rangle \in W(\text{wsn}_i(\tau))$, then at time τ p_i has already bscd-delivered and processe a message $\langle j, snj', \text{WRITE}(v')\rangle$ with $snj' \geq snj$. Since BSCD-broadcast preserves sequence number ordering, p_i has already bscd-delivered $\langle j, snj, \text{WRITE}(v)\rangle$, therefore $\langle j, snj, \text{WRITE}(v)\rangle \in U_i(\tau)$.

The two equivalency relations follow directly from replacing $U_i(\tau) \cap W$ and $U_j(\tau') \cap W$ respectively by $W(\text{wsn}_i(\tau))$ and $W(\text{wsn}_j(\tau'))$ and applying the definitions. ◀

▶ **Definition 39.** *If $op = \langle i, sni, x\rangle \in O$ then let $\text{sna}(op)$, $\text{witness}(op)$ and $\tau(op)$ be defined as follows:*
- *if $x =$SYNC(), then by definition of O, p_i is a correct process. Let:*
  - $\text{sna}(op)$ *be the value of wsn_i when p_i returns $reg_i[1 \dots n]$ (line 2),*
  - $\text{witness}(op) = p_i$,
  - *$\tau(op)$ be the time at which p_i executes line 7 for the last time before returning at line 2.*
- *if $x =$WRITE(v) then let:*
  - $\text{sna}(op)$ *be the smallest sequence number array observed by a correct process and that has registered the write operation op, which exists because $<_{sna}$ is a well-founded total order on WSN (by Lemma 30):*

$$\text{sna}(op) = \min\{sna \in WSN : sna[i] \geqslant sni\},$$

- *$\tau(op)$ be the smallest time at which a correct process p_j executed line 7 with $wsn_j = \text{sna}(op)$,*
- *$\text{witness}(op) = p_j$ be the correct process that executed line 7 with $wsn_j = \text{sna}(op)$ at $\tau(op)$.*

In both cases, the following relations hold:

$$\begin{aligned} \text{sna}(op) &= \text{wsn}_{\text{witness}(op)}(\tau(op)), \\ op &\in U_{\text{witness}(op)}(\tau(op)). \end{aligned}$$

And if p_i is a correct process that invoked op at $\tau_{invoke}(op)$ and returned at $\tau_{return}(op)$ then:

$$\tau_{invoke}(op) < \tau(op) < \tau_{return}(op).$$

▶ **Lemma 40.** *Let op and op' be two distinct operations such that $op \prec op'$. We have $\text{sna}(op) \leqslant \text{sna}(op')$. Moreover, if $op' \in W$ then $\text{sna}(op) < \text{sna}(op')$.*

Proof. op and op' are invoked by correct processes. We have $\tau(op) < \tau_{return}(op) < \tau_{invoke}(op') < \tau(op')$.

By BSCD-order, we have:

$$U_{\text{witness}(op)}(\tau(op)) \subseteq U_{\text{witness}(op')}(\tau(op')) \quad \vee \quad U_{\text{witness}(op')}(\tau(op')) \subseteq U_{\text{witness}(op)}(\tau(op)).$$

However op' was bscd-broadcast after $\tau(op)$ therefore $\text{witness}(op)$ cannot have bscd-delivered op' at time $\tau(op)$. Therefore: $op' \in U_{\text{witness}(op')}(\tau(op')) \backslash U_{\text{witness}(op)}(\tau(op))$.

Therefore we are in the case: $U_{\text{witness}(op)}(\tau(op)) \subset U_{\text{witness}(op')}(\tau(op'))$, from which we deduce:

$$\begin{aligned} U_{\text{witness}(op)}(\tau(op)) \cap W &\subseteq U_{\text{witness}(op')}(\tau(op')) \cap W, \\ \text{wsn}_{\text{witness}(op)}(\tau(op)) &\leqslant \text{wsn}_{\text{witness}(op')}(\tau(op')), \\ \text{sna}(op) &\leqslant \text{sna}(op'). \end{aligned}$$

Moreover, if $op' \in W$ then the inclusion and inequalities become strict and we have the desired results. ◀

▶ **Definition 41.** *Let $op, op' \in O$ be two distinct operations. We define $\rightarrow$ as $op \rightarrow op'$ if either one of the following holds:*

- *(A) $op \prec op'$,*
- *(B) $\text{sna}(op) < \text{sna}(op')$,*
- *(C) $(\text{sna}(op) = \text{sna}(op')) \wedge (op \in W) \wedge (op' \notin W)$.*

Let $\rightarrow^*$ be the reflexive transitive closure of $\rightarrow$.

▶ **Lemma 42.** *$\rightarrow^*$ is a partial order on C.*

Proof. $\rightarrow^*$ is reflexive transitive by construction. Let us prove antisymetry:

Suppose we have $op \neq op'$ such that $op \rightarrow^* op'$ and $op' \rightarrow^* op$. By definition of $\rightarrow^*$ this means there are $op_0, op_1, op_2, \ldots, op_m$ for an $m > 1$ such that $op_0 = op_m = op$, $op_k = op'$ for some $k \in [1, m-1]$, and $op_i \rightarrow op_{i+1}$ for all $i < m$.

By the definition and Lemma 40, we have that $\forall i < m, \text{sna}(op_i) \leqslant \text{sna}(op_{i+1})$.

Moreover $\text{sna}(op_0) = \text{sna}(op_m)$, and $\leqslant$ on sequence number arrays is antisymmeteric, thus we have: $\forall i < m, \text{sna}(op_i) = \text{sna}(op_0)$, which eliminates case (B) for all i.

We have two cases:

- If for any $i < m$, $op_i \notin W$, then we are necessarily in case (A): $op_i \prec op_{i+1}$. Since $\text{sna}(op_i) = \text{sna}(op_{i+1})$, by the contrapositive of Lemma 40, we have $op_{i+1} \notin W$.
  By applying this same reasonning recursively, we have that $\forall i < m, op_i \prec op_{i+1}$. As $\prec$ is included in the total order of time, it is a partial order, therefore antisymmetric. We have a contradiction.
- Otherwise, $\forall i < m, op_i \in W$, which eliminates case (C). By the contrapositive of Lemma 40, since $\text{sna}(op_i) = \text{sna}(op_{i+1})$, we have $\forall i, op_i \not\prec op_{i+1}$. All three cases for $op_i \to op_{i+1}$ are impossible, we have a contradiction.

In both cases we have shown a contradiction, therefore $\to^*$ is antisymmetric. ◀

Final proof of Theorem 23

Proof. Let $\to_{op}$ be a topological sort of $\to^*$. $\to_{op}$ is a total order that extends $\to^*$. $\to_{op}$ includes the process order $\prec$ therefore it is real-time compliant.

Let us consider a snapshot operation $op = \langle i, sni, \textsc{sync}() \rangle$ by a correct process p_i, and let us show that the value read by p_i in each register is the last value written to that register before op according to $\to_{op}$.

Let us consider a register k and let $snk = \text{sna}(op)[k]$. According to line 3, the value returned by op for register k is a value v such that $op_k = \langle k, snk, \textsc{write}(v) \rangle$ is the last write operation on k known by p_i when the snapshot operation returns. By definition of $\text{sna}(op)$ we have $\text{sna}(op_k) \leqslant \text{sna}(op)$, moreover op_k is a write and op is a read, therefore $op_k \to_{op} op$. Moreover, for any different write operation $op'_k = \langle k, \text{snk}', \textsc{write}(v') \rangle$ on register k, we have $snk \neq snk'$. If $snk' < snk$ then $op'_k \to_{op} op_k$. Otherwise $snk' > snk$, in which case $\text{sna}(op'_k) > \text{sna}(op)$, therefore $op \to_{op} op'_k$. In both cases, v is the last value written on register k before the snapshot operation op according to $\to_{op}$.

We have thus shown that Algorithm 3 builds a linearizable single-writer multi-reader Byzantine tolerant snapshot object. ◀

Asymmetric Distributed Trust

Christian Cachin
University of Bern, Switzerland
https://crypto.unibe.ch/cc/
cachin@inf.unibe.ch

Björn Tackmann
DFINITY Foundation, Zürich, Switzerland
bjoern@dfinity.org

Abstract

Quorum systems are a key abstraction in distributed fault-tolerant computing for capturing trust assumptions. They can be found at the core of many algorithms for implementing reliable broadcasts, shared memory, consensus and other problems. This paper introduces *asymmetric Byzantine quorum systems* that model subjective trust. Every process is free to choose which combinations of other processes it trusts and which ones it considers faulty. Asymmetric quorum systems strictly generalize standard Byzantine quorum systems, which have only one global trust assumption for all processes. This work also presents protocols that implement abstractions of shared memory and broadcast primitives with processes prone to Byzantine faults and asymmetric trust. The model and protocols pave the way for realizing more elaborate algorithms with asymmetric trust.

2012 ACM Subject Classification Theory of computation → Cryptographic protocols; Software and its engineering → Distributed systems organizing principles

Keywords and phrases Quorums, consensus, distributed trust, blockchains, cryptocurrencies

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.7

Related Version A complete version of the paper is available at http://arxiv.org/abs/1906.09314.

Funding This work has been supported in part by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 780477 PRIViLEDGE.

Acknowledgements The authors thank Orestis Alpos, Sabine Brunner, Marko Vukolić, and Luca Zanolini for interesting discussions. Work done while both authors were at IBM Research - Zurich.

1 Introduction

Byzantine quorum systems [21] are a fundamental primitive for building resilient distributed systems from untrusted components. Given a set of nodes, a quorum system captures a trust assumption on the nodes in terms of potentially malicious protocol participants and colluding groups of nodes. Based on quorum systems, many well-known algorithms for *reliable broadcast*, *shared memory*, *consensus* and more have been implemented; these are the main abstractions to synchronize the correct nodes with each other and to achieve consistency despite the actions of the faulty, so-called *Byzantine* nodes.

Traditionally, trust in a Byzantine quorum system for a set of processes $\mathcal{P}$ has been *symmetric*. In other words, a global assumption specifies which processes may fail, such as the simple and prominent *threshold quorum* assumption, in which any subset of $\mathcal{P}$ of a given maximum size may collude and act against the protocol. The most basic threshold Byzantine quorum system, for example, allows all subsets of up to $f < n/3$ processes to fail. Some classic works also model arbitrary, non-threshold symmetric quorum systems [21, 15], but these have not actually been used in practice.

© Christian Cachin and Björn Tackmann;
licensed under Creative Commons License CC-BY
23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller; Article No. 7; pp. 7:1–7:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

However, trust is inherently subjective. *De gustibus non est disputandum.*[1] Estimating which processes will function correctly and which ones will misbehave may depend on personal taste. A myriad of local choices influences one process' trust in others, especially because there are so many forms of "malicious" behavior. Some processes might not even be aware of all others, yet a process should not depend on unknown third parties in a distributed collaboration. How can one model asymmetric trust in distributed protocols? Can traditional Byzantine quorum systems be extended to subjective failure assumptions? How do the standard protocols generalize to this model?

In this paper, we answer these questions and introduce models and protocols for asymmetric distributed trust. We formalize *asymmetric quorum systems* for asynchronous protocols, in which every process can make its own assumptions about Byzantine faults of others. We introduce several protocols with asymmetric trust that strictly generalize the existing algorithms, which require common trust.

Our formalization takes up earlier work by Damgård et al. [10] and starts out with the notion of a fail-prone system that forms the basis of a symmetric Byzantine quorum system. A global fail-prone system for a process set $\mathcal{P}$ contains all maximal subsets of $\mathcal{P}$ that might jointly fail during an execution. In an asymmetric quorum system, every process specifies its *own* fail-prone system and a corresponding set of local quorums. These local quorum systems satisfy a *consistency condition* that ranges across all processes and a local *availability condition*, and generalize symmetric Byzantine quorum system according to Malkhi and Reiter [21].

Interest in consensus protocols based on Byzantine quorum systems has surged recently because of their application to permissioned blockchain networks [6, 1]. Typically run by a consortium, such distributed ledgers often use *Byzantine-fault tolerant (BFT)* protocols like PBFT [7] for consensus that rely on symmetric threshold quorum systems. The Bitcoin blockchain and many other cryptocurrencies, which triggered this development, started from different assumptions and use so-called permissionless protocols, in which everyone may participate. Those algorithms capture the relative influence of the participants on consensus decisions by an external factor, such as "proof-of-work" or "proof-of-stake."

A middle ground between permissionless blockchains and BFT-based ones has been introduced by the blockchain networks of Ripple (`https://ripple.com`) and Stellar (`https://stellar.org`). Their stated model for achieving network-level consensus uses subjective trust in the sense that each process declares a local list of processes that it "trusts" in the protocol.

Consensus in the *Ripple* blockchain (and for the *XRP* cryptocurrency on the *XRP Ledger*) is executed by its validator nodes. Each validator declares a *Unique Node List (UNL)*, which are nodes that a given participant trusts. Questions have been raised about the kind of decentralization offered by the Ripple protocol. *Stellar* was created as an evolution of Ripple that shares much of the same design philosophy. The Stellar consensus protocol [22] powers the *Stellar Lumen (XLM)* cryptocurrency and introduces *federated Byzantine quorum systems (FBQS)*; these bear superficial resemblance with our asymmetric quorum systems but differ technically. However, standard Byzantine quorum systems and FBQS are *not* comparable because (1) an FBQS when instantiated with the same trust assumption for all processes does not reduce to a symmetric quorum system and (2) existing protocols do not generalize to FBQS.

[1] There is no disputing about taste.

Understanding how such ideas of subjective trust, as manifested in the Ripple and Stellar blockchains, relate to traditional quorum systems is the main motivation for this work. Our protocols for asymmetric trust generalize the well-known, classic algorithms in the literature and therefore look superficially similar. This should be seen as a feature, actually, because simplicity and modularity are important guiding principles in science.

Our contributions are as follows:

- We introduce asymmetric Byzantine quorum systems formally in Section 4 as an extension of standard Byzantine quorum systems and discuss some of their properties.
- In Section 5, we show two implementations of a shared register, with single-writer, multi-reader regular semantics, using asymmetric Byzantine quorum systems.
- We examine broadcast primitives in the Byzantine model with asymmetric trust in Section 6. In particular, we define and implement Byzantine consistent and reliable broadcast protocols. The latter primitive is related to a "federated voting" protocol used by Stellar consensus [22].

The long version of the paper contains more details and all proofs [5].

2 Related work

Damgård et al. [10] introduce some basics of asymmetric trust in the context of synchronous protocols for secure distributed computation by modeling process-specific fail-prone systems. They state the consistency property of asymmetric Byzantine quorums but do not prove that the B^3 property is required.

The *Ripple* consensus protocol is run by an open set of validator nodes. The protocol uses votes, similar to standard consensus protocols, whereby each validator only communicates with the validators in its UNL. Each validator chooses its own UNL, which makes it possible for anyone to participate, in principle, similar to proof-of-work blockchains. Early investigations suggested that the intersection of the UNLs of every two validators should be at least 20% of each list [25], assuming that also less than one fifth of the validators in the UNL of every node might be faulty. An independent analysis by Armknecht et al. [2] later argued that this bound must be more than 40%. A recent technical report of Chase and MacBrough [9, Thm. 8] concludes, under the same assumption of $f < n/5$ faulty nodes in every UNL of size n, that the UNL overlap should actually be at least 90%.

The *Stellar consensus protocol (SCP)* also features open membership and lets every node express its own set of trusted nodes [22]. Generalizing from Ripple's flat lists of unique nodes, every node declares a collection of trusted sets called *quorum slices*, whereby a slice is "the subset of a quorum convincing one particular node of agreement." A *quorum* in Stellar is a set of nodes "sufficient to reach agreement," defined as a set of nodes that contains one slice for each member node. The quorum choices of all nodes together yield a *federated Byzantine quorum systems (FBQS)*. The Stellar white paper states properties of FBQS and protocols that build on them. A recent paper of García-Pérez and Gotsman [12] elaborates on FBQS. Recent papers further describe Stellar's FBQS [18] and a closely related notion called *personal Byzantine quorum systems* [19].

However, these concepts and protocols do not map to known primitives in distributed computing. The FBQS notion is at odds with the usual notion of a Byzantine quorum system in the sense that it does not reduce to a symmetric quorum system for symmetric trust choices.

In contrast to Ripple's and Stellar's attempts to formalize subjective trust, our asymmetric quorum formulation extends the well-established quorum systems that underlie classic

Byzantine consensus. We also introduce several protocols with asymmetric trust that strictly generalize existing standard algorithms, which have so far required common trust and knowledge of all nodes.

3 System model

We consider a system of n *processes* $\mathcal{P} = \{p_1, \ldots, p_n\}$ that communicate with each other. The processes interact asynchronously with each other through exchanging messages. The system itself is asynchronous, i.e., the delivery of messages among processes may be delayed arbitrarily and the processes have no synchronized clocks. Every process is identified by a name, but such identifiers are not made explicit. We use standard notions of protocols, events, functionalities, and fair executions from the literature [20, 4]. All processes are linked by reliable and authenticated point-to-point FIFO channels [14, 4].

A process that follows its protocol during an execution is called *correct*. On the other hand, a *faulty* process may crash or even deviate arbitrarily from its specification, e.g., when *corrupted* by an adversary; such processes are also called *Byzantine*. We consider only Byzantine faults here and assume for simplicity that the faulty processes fail right at the start of an execution.

Some protocols use *digital signatures*, for which we adopt an idealized implementation with two operations, $sign_i$ and $verify_i$. It follows the standard semantics; details appear in the long version [5].

4 Asymmetric Byzantine quorum systems

4.1 Symmetric trust

Quorum systems are well-known in settings with symmetric trust. As demonstrated by many applications to distributed systems, ordinary quorum systems [23] and Byzantine quorum systems [21] play a crucial role in formulating resilient protocols that tolerate faults through replication [8]. A quorum system typically ensures a consistency property among the processes in an execution, despite the presence of some faulty processes.

For the model with Byzantine faults, *Byzantine quorum systems* have been introduced by Malkhi and Reiter [21]. This notion is defined with respect to a *fail-prone system* $\mathcal{F} \subseteq 2^{\mathcal{P}}$, a collection of subsets of $\mathcal{P}$, none of which is contained in another, such that some $F \in \mathcal{F}$ with $F \subseteq \mathcal{P}$ is called a *fail-prone set* and contains all processes that may at most fail together in some execution [21]. A fail-prone system is the same as the *basis* of an *adversary structure*, which was introduced independently by Hirt and Maurer [15].

A fail-prone system captures an assumption on the possible failure patterns that may occur. It specifies all maximal sets of faulty processes that a protocol should tolerate in an execution; this means that a protocol designed for $\mathcal{F}$ achieves its properties as long as the set F of actually faulty processes satisfies $F \in \mathcal{F}^*$. Here and from now on, the notation $\mathcal{A}^*$ for a system $\mathcal{A} \subseteq 2^{\mathcal{P}}$, denotes the collection of all subsets of the sets in $\mathcal{A}$, that is,

▶ **Definition 1** (Byzantine quorum system [21]). *A* Byzantine quorum system *for $\mathcal{F}$ is a collection of sets of processes $\mathcal{Q} \subseteq 2^{\mathcal{P}}$, where each $Q \in \mathcal{Q}$ is called a* quorum*, such that the following properties hold:*

Consistency: *The intersection of any two quorums contains at least one process that is not faulty, i.e.,*

$$\forall Q_1, Q_2 \in \mathcal{Q}, \forall F \in \mathcal{F} : Q_1 \cap Q_2 \not\subseteq F.$$

Availability: *For any set of processes that may fail together, there exists a disjoint quorum in $\mathcal{Q}$, i.e.,*

$$\forall F \in \mathcal{F} : \exists Q \in \mathcal{Q} : F \cap Q = \emptyset.$$

The above notion is also known as a *Byzantine dissemination quorum system* [21] and allows a protocol to be designed despite arbitrary behavior of the potentially faulty processes. The notion generalizes the usual threshold failure assumption for Byzantine faults [24], which considers that any set of f processes are equally likely to fail.

We say that a set system $\mathcal{T}$ *dominates* another set system $\mathcal{S}$ if for each $S \in \mathcal{S}$ there is some $T \in \mathcal{T}$ such that $S \subseteq T$ [11]. In this sense, a quorum system for $\mathcal{F}$ is *minimal* whenever it does not dominate any other quorum system for $\mathcal{F}$.

Similarly to the threshold case, where $n > 3f$ processes overall are needed to tolerate f faulty ones in many Byzantine protocols, Byzantine quorum systems can only exist if not "too many" processes fail.

▶ **Definition 2** (Q^3-condition [21, 15]). *A fail-prone system $\mathcal{F}$ satisfies the Q^3-condition, abbreviated as $Q^3(\mathcal{F})$, whenever it holds*

$$\forall F_1, F_2, F_3 \in \mathcal{F} : \mathcal{P} \not\subseteq F_1 \cup F_2 \cup F_3.$$

In other words, $Q^3(\mathcal{F})$ means that no *three* fail-prone sets together cover the whole system of processes. A Q^k-condition can be defined like this for any $k \geq 2$ [15].

The following lemma considers the *bijective complement* of a process set $\mathcal{S} \subseteq 2^{\mathcal{P}}$, which is defined as $\overline{\mathcal{S}} = \{\mathcal{P} \setminus S | S \in \mathcal{S}\}$, and turns $\mathcal{F}$ into a Byzantine quorum system.

▶ **Lemma 3** ([21, Theorem 5.4]). *Given a fail-prone system $\mathcal{F}$, a Byzantine quorum system for $\mathcal{F}$ exists if and only if $Q^3(\mathcal{F})$. In particular, if $Q^3(\mathcal{F})$ holds, then $\overline{\mathcal{F}}$, the bijective complement of $\mathcal{F}$, is a Byzantine quorum system.*

The quorum system $\mathcal{Q} = \overline{\mathcal{F}}$ is called the *canonical quorum system* of $\mathcal{F}$. According to the duality between $\mathcal{Q}$ and $\mathcal{F}$, properties of $\mathcal{F}$ are often ascribed to $\mathcal{Q}$ as well; for instance, we say $Q^3(\mathcal{Q})$ holds if and only if $Q^3(\mathcal{F})$. However, note that the canonical quorum system is not always minimal. For instance, if $\mathcal{F}$ consists of all sets of $f \ll n/3$ processes, then each quorum in the canonical quorum system has $n - f$ members, but also the family of all subsets of $\mathcal{P}$ with $\lceil \frac{n+f+1}{2} \rceil < n - f$ processes forms a quorum system.

Core sets. A *core set* C for $\mathcal{F}$ is a minimal set of processes that contains at least one correct process in every execution. More precisely, $C \subseteq \mathcal{P}$ is a core set whenever (1) for all $F \in \mathcal{F}$, it holds $\mathcal{P} \setminus F \cap C \neq \emptyset$ (and, equivalently, $C \not\subseteq F$) and (2) for all $C' \subsetneq C$, there exists $F \in \mathcal{F}$ such that $\mathcal{P} \setminus F \cap C' = \emptyset$ (and, equivalently, $C' \subseteq F$). With the threshold failure assumption, every set of $f + 1$ processes is a core set. A *core set system* $\mathcal{C}$ is the minimal collection of all core sets, in the sense that no set in $\mathcal{C}$ is contained in another. Core sets can be complemented by *survivor sets*, as shown by Junqueira et al. [16]. This yields a dual characterization of resilient distributed protocols, which parallels ours using fail-prone sets and quorums.

4.2 Asymmetric trust

In our model with asymmetric trust, every process is free to make its own trust assumption and to express this with a fail-prone system. Hence, an *asymmetric fail-prone system* $\mathbb{F} = [\mathcal{F}_1, \ldots, \mathcal{F}_n]$ consists of an array of fail-prone systems, where $\mathcal{F}_i$ denotes the trust assumption of p_i. One often assumes $p_i \notin F_i$ for practical reasons, but this is not necessary. This notion has earlier been formalized by Damgård et al. [10].

▶ **Definition 4** (Asymmetric Byzantine quorum system). *An* asymmetric Byzantine quorum system *for* $\mathbb{F}$ *is an array of collections of sets* $\mathbb{Q} = [\mathcal{Q}_1, \ldots, \mathcal{Q}_n]$*, where* $\mathcal{Q}_i \subseteq 2^{\mathcal{P}}$ *for* $i \in [1, n]$*. The set* $\mathcal{Q}_i \subseteq 2^{\mathcal{P}}$ *is called the* quorum system of p_i *and any set* $Q_i \in \mathcal{Q}_i$ *is called a* quorum (set) for p_i*. It satisfies:*

Consistency: *The intersection of two quorums for any two processes contains at least one process for which either process assumes that it is not faulty, i.e.,*

$$\forall i, j \in [1, n], \forall Q_i \in \mathcal{Q}_i, \forall Q_j \in \mathcal{Q}_j, \forall F_{ij} \in {\mathcal{F}_i}^* \cap {\mathcal{F}_j}^* : Q_i \cap Q_j \not\subseteq F_{ij}.$$

Availability: *For any process* p_i *and any set of processes that may fail together according to* p_i*, there exists a disjoint quorum for* p_i *in* $\mathcal{Q}_i$*, i.e.,*

$$\forall i \in [1, n], \forall F_i \in \mathcal{F}_i : \exists Q_i \in \mathcal{Q}_i : F_i \cap Q_i = \emptyset.$$

The existence of asymmetric quorum systems can be characterized with a property that generalizes the Q^3-condition for the underlying asymmetric fail-prone systems as follows.

▶ **Definition 5** (B^3-condition). *An asymmetric fail-prone system* $\mathbb{F}$ *satisfies the* B^3-condition*, abbreviated as* $B^3(\mathbb{F})$*, whenever it holds that*

$$\forall i, j \in [1, n], \forall F_i \in \mathcal{F}_i, \forall F_j \in \mathcal{F}_j, \forall F_{ij} \in {\mathcal{F}_i}^* \cap {\mathcal{F}_j}^* : \mathcal{P} \not\subseteq F_i \cup F_j \cup F_{ij}$$

The following result is the generalization of Lemma 3 for asymmetric quorum systems; it was stated by Damgård et al. [10] without proof. As for symmetric quorum systems, we use this result and say that $B^3(\mathbb{Q})$ holds whenever the asymmetric $\mathbb{Q}$ consists of the canonical quorum systems for $\mathbb{F}$ and $B^3(\mathbb{F})$ holds. A proof of the following result appears in the long version [5].

▶ **Theorem 6.** *An asymmetric fail-prone system* $\mathbb{F}$ *satisfies* $B^3(\mathbb{F})$ *if and only if there exists an asymmetric quorum system for* $\mathbb{F}$*.*

Kernels. Given a symmetric Byzantine quorum system $\mathcal{Q}$, we define a *kernel* K as a set of processes that overlaps with every quorum and that is minimal in this respect. Formally, $K \subseteq \mathcal{P}$ is a *kernel of* $\mathcal{Q}$ if and only if

$$\forall Q \in \mathcal{Q} : K \cap Q \neq \emptyset$$

and

$$\forall K' \subsetneq K : \exists Q \in \mathcal{Q} : K \cap Q = \emptyset.$$

The *kernel system* $\mathcal{K}$ of $\mathcal{Q}$ is the set of all kernels of $\mathcal{Q}$.

For example, under a threshold failure assumption where any f processes may fail and the quorums are all sets of $\lceil \frac{n+f+1}{2} \rceil$ processes, every set of $\lfloor \frac{n-f+1}{2} \rfloor$ processes is a kernel.

The definition of a kernel is related to that of a core set in the following sense. For a given maximal fail-prone system $\mathcal{F}$, consider its canonical quorum system $\mathcal{Q} = \overline{\mathcal{F}}$; if $\mathcal{Q}$ is minimal, then the kernel system of $\mathcal{Q}$ is the same as the core-set system for $\mathcal{F}$.

Asymmetric core sets and kernels. Let $\mathbb{F} = [\mathcal{F}_1, \ldots, \mathcal{F}_n]$ be an asymmetric fail-prone system. An *asymmetric core set system* $\mathbb{C}$ is an array of collections of sets $[\mathcal{C}_1, \ldots, \mathcal{C}_n]$ such that each $\mathcal{C}_i$ is a core set system for the fail-prone system $\mathcal{F}_i$. We call a set $C_i \in \mathcal{C}_i$ a *core set for* p_i.

Given an asymmetric quorum system $\mathbb{Q}$ for $\mathbb{F}$, an *asymmetric kernel system* for $\mathbb{Q}$ is defined analogously as the array $\mathbb{K} = [\mathcal{K}_1, \ldots, \mathcal{K}_n]$ that consists of the kernel systems for all processes in $\mathcal{P}$ with respect to $\mathbb{Q}$; a set $K_i \in \mathcal{K}_i$ is called a *kernel for* p_i.

Naïve and wise processes. The faults or corruptions occurring in a protocol execution with an underlying quorum system imply a set F of actually *faulty processes*. However, no process knows F and this information is only available to an observer outside the system. With a traditional quorum system $\mathcal{Q}$ designed for a fail-prone set $\mathcal{F}$, the guarantees of a protocol usually hold as long as $F \in \mathcal{F}$. Recall that such protocol properties apply to *correct* processes only but not to faulty ones.

With asymmetric quorums, we further distinguish between two kinds of correct processes, depending on whether they considered F in their trust assumption or not. Given a protocol execution, the processes are therefore partitioned into three types:

Faulty: A process $p_i \in F$ is *faulty*.

Naïve: A correct process p_i for which $F \notin {\mathcal{F}_i}^*$ is called *naïve*.

Wise: A correct process p_i for which $F \in {\mathcal{F}_i}^*$ is called *wise*.

The naïve processes are new for the asymmetric case, as all processes are wise under a symmetric trust assumption. Protocols for asymmetric quorums cannot guarantee the same properties for naïve processes as for wise ones, since the naïve processes may have the "wrong friends."

Guilds. If too many processes are naïve or even fail during a protocol run with asymmetric quorums, then protocol properties cannot be ensured. A *guild* is a set of wise processes that contains at least one quorum for each member; by definition this quorum consists only of wise processes. A guild ensures liveness and consistency for typical protocols. This generalizes from protocols for symmetric quorum systems, where the correct processes in every execution form a quorum by definition. (A guild represents a group of influential and well-connected wise processes, like in the real world.)

▶ **Definition 7** (Guild). *Given a fail-prone system $\mathbb{F}$, an asymmetric quorum system $\mathbb{Q}$ for $\mathbb{F}$, and a protocol execution with faulty processes F, a* guild $\mathcal{G}$ for F *satisfies two properties:*
Wisdom: *$\mathcal{G}$ is a set of wise processes:*

$$\forall p_i \in \mathcal{G} : F \in {\mathcal{F}_i}^*.$$

Closure: *$\mathcal{G}$ contains a quorum for each of its members:*

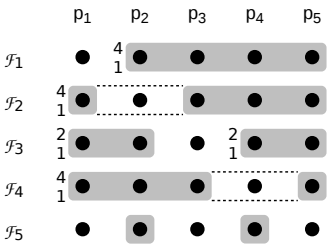
$$\forall p_i \in \mathcal{G} : \exists Q_i \in \mathcal{Q}_i : Q_i \subseteq \mathcal{G}.$$

Superficially a guild seems similar to a "quorum" in the Stellar consensus protocol [22], but the two notions actually differ because a guild contains only wise processes and Stellar's quorums do not distinguish between naïve and wise processes.

Observe that for a specific execution, the union of two guilds is again a guild, since the union consists only of wise processes and contains again a quorum for each member. Hence, every execution with a guild contains a unique *maximal guild* $\mathcal{G}_{\max}$.

Example. We define an example asymmetric fail-prone system $\mathbb{F}_A$ on $\mathcal{P} = \{p_1, p_2, p_3, p_4, p_5\}$. The notation $\Theta_k^n(\mathcal{S})$ for a set $\mathcal{S}$ with n elements denotes the "threshold" combination operator and enumerates all subsets of $\mathcal{S}$ of cardinality k. W.l.o.g. every process trusts itself. The diagram below shows fail-prone sets as shaded areas and the notation $\frac{n}{k}$ in front of a fail-prone set stands for k out of the n processes in the set.

$$\mathbb{F}_A: \quad \begin{aligned} \mathcal{F}_1 &= \Theta_1^4(\{p_2, p_3, p_4, p_5\}) \\ \mathcal{F}_2 &= \Theta_1^4(\{p_1, p_3, p_4, p_5\}) \\ \mathcal{F}_3 &= \Theta_1^2(\{p_1, p_2\}) * \Theta_1^2(\{p_4, p_5\}) \\ \mathcal{F}_4 &= \Theta_1^4(\{p_1, p_2, p_3, p_5\}) \\ \mathcal{F}_5 &= \{\{p_2, p_4\}\} \end{aligned}$$



The operator $*$ for two sets satisfies $\mathcal{A} * \mathcal{B} = \{A \cup B | A \in \mathcal{A}, B \in \mathcal{B}\}$. As one can verify in a straightforward way, $B^3(\mathbb{F}_A)$ holds. Let $\mathbb{Q}_A$ be the canonical asymmetric quorum system for $\mathbb{F}_A$. Note that since $\mathbb{F}_A$ contains the fail-prone systems of p_3 and p_5 that permit two faulty processes each, this fail-prone system cannot be obtained as a special case of $\Theta_1^5(\{p_1, p_2, p_3, p_4, p_5\})$. When $F = \{p_2, p_4\}$, for example, then processes p_3 and p_5 are wise and p_1 is naïve.

5 Shared memory

This section illustrates a first application of asymmetric quorum systems: how to emulate shared memory, represented by a *register*. Maintaining a shared register reliably in a distributed system subject to faults is perhaps the most fundamental task for which ordinary, symmetric quorum systems have been introduced, in the models with crashes [13] and with Byzantine faults [21]. We present definitions and one protocol for implementing a register with asymmetric quorums in this section.

The long version [5] also presents a second protocol without digital signatures and explains why federated Byzantine quorum systems according to Stellar [22] fail to directly emulate shared memory. This protocol also illustrates the role of an asymmetric core set system that generalizes the notion of an $(f + 1)$-process subset in the threshold model.

5.1 Definitions

We use the standard notions for operations and their precedence to formalize a register as a shared object. More precisely, a *register* with domain $\mathcal{X}$ provides two operations: *write*(x), which is parameterized by a value $x \in \mathcal{X}$ and outputs a token ACK when it completes; and *read*, which takes no parameter for invocation but outputs a value $x \in \mathcal{X}$ upon completion.

We consider a *single-writer* (or *SW*) register, where only a designated process p_w may invoke *write*, and permit *multiple readers* (or *MR*), that is, every process may execute a *read* operation. The register is initialized with a special value x_0, which is written by an imaginary *write* operation that occurs before any process invokes operations. We consider *regular* semantics under concurrent access [17]; the extension to other forms of concurrent memory, including an atomic register, proceeds analogously.

It is customary in the literature to assume that the writer and reader processes are correct; with asymmetric quorums we assume explicitly that readers and writers are *wise*. We illustrate below why one cannot extend the guarantees of the register to naïve processes.

▶ **Definition 8** (Asymmetric Byzantine SWMR regular register). *A protocol emulating an* asymmetric SWMR regular register *satisfies:*

Liveness: *If a wise process p invokes an operation on the register, p eventually completes the operation.*

Safety: *Every read operation of a wise process that is not concurrent with a write returns the value written by the most recent, preceding write of a wise process; furthermore, a read operation of a wise process concurrent with a write of a wise process may also return the value that is written concurrently.*

5.2 Protocol with authenticated data

In Algorithm 1, we describe a protocol for emulating a regular SWMR register with an asymmetric Byzantine quorum system, for a designated writer p_w and a reader $p_r \in \mathcal{P}$. The protocol uses *data authentication* implemented with digital signatures. This protocol is the same as the classic one of Malkhi and Reiter [21] that uses a Byzantine dissemination quorum system and where processes send messages to each other over point-to-point links. The difference lies in the individual choices of quorums by the processes and that it ensures safety and liveness for wise processes.

In the register emulation, the writer p_w obtains ACK messages from all processes in a quorum $Q_w \in \mathcal{Q}_w$; likewise, the reader p_r waits for a VALUE message carrying a value/-timestamp pair from every process in a quorum $Q_r \in \mathcal{Q}_r$ of the reader.

Algorithm 1 Emulation of an asymmetric SWMR regular register (process p_i).

State
- wts: sequence number of write operations, stored only by writer p_w
- rid: identifier of read operations, used only by reader
- ts, v, σ: current state stored by p_i: timestamp, value, signature

upon invocation $write(v)$ **do** // only if p_i is writer p_w
- $wts \leftarrow wts + 1$
- $\sigma \leftarrow sign_w(\text{WRITE}\|w\|wts\|v)$
- send message $[\text{WRITE}, wts, v, \sigma]$ to all $p_j \in \mathcal{P}$
- **wait for** receiving a message [ACK] from all processes in some quorum $Q_w \in \mathcal{Q}_w$

upon invocation $read$ **do** // only if p_i is reader p_r
- $rid \leftarrow rid + 1$
- send message $[\text{READ}, rid]$ to all $p_j \in \mathcal{P}$
- **wait for** receiving messages $[\text{VALUE}, r_j, ts_j, v_j, \sigma_j]$ from all processes in some $Q_r \in \mathcal{Q}_r$ **such that** $r_j = rid$ **and** $verify_w(\sigma_j, \text{WRITE}\|w\|ts\|v_j)$
- **return** $highestval(\{(ts_j, v_j) | j \in Q_r\})$

upon receiving a message $[\text{WRITE}, ts', v', \sigma']$ from p_w **do** // every process
- **if** $ts' > ts$ **then**
  - $(ts, v, \sigma) \leftarrow (ts', v', \sigma')$
- send message [ACK] to p_w

upon receiving a message $[\text{READ}, r]$ from p_r **do** // every process
- send message $[\text{VALUE}, r, ts, v, \sigma]$ to p_r

The function $highestval(S)$ takes a set of timestamp/value pairs S as input and outputs the value in the pair with the largest timestamp, i.e., v such that $(ts, v) \in S$ and $\forall (ts', v') \in S : ts' < ts \vee (ts', v') = (ts, v)$. Note that this v is unique in Algorithm 1 because p_w is correct. The protocol uses digital signatures, modeled by operations $sign_i$ and $verify_i$, as introduced earlier.

▶ **Theorem 9.** *Algorithm 1 emulates an asymmetric Byzantine SWMR regular register.*

Example. We show why the guarantees of this protocol with asymmetric quorums hold only for wise readers and writers. Consider $\mathbb{Q}_A$ from the last section and an execution in which p_2 and p_4 are faulty, and therefore p_1 is naïve and p_3 and p_5 are wise. A quorum for p_1 consists of p_1 and three processes in $\{p_2, \ldots, p_5\}$; moreover, every process set that contains p_3, one of $\{p_1, p_2\}$ and one of $\{p_4, p_5\}$ is a quorum for p_3.

We illustrate that if naïve p_1 writes, then a wise reader p_3 may violate safety. Suppose that all correct processes, especially p_3, store timestamp/value/signature triples from an operation that has terminated and that wrote x. When p_1 invokes *write*(u), it obtains [ACK] messages from all processes except p_3. This is a quorum for p_1. Then p_3 runs a *read* operation and receives the outdated values representing x from itself (p_3 is correct but has not been involved in writing u) and also from the faulty p_2 and p_4. Hence, p_3 outputs x instead of u.

Analogously, with the same setup of every process initially storing a representation of x but with wise p_3 as writer, suppose p_3 executes *write*(u). It obtains [ACK] messages from p_2, p_3, and p_4 and terminates. When p_1 subsequently invokes *read* and receives values representing x, from correct p_1 and p_5 and from faulty p_2 and p_4, then p_1 outputs x instead of y and violates safety as a naïve reader.

Since the sample operations are not concurrent, the implication actually holds also for registers with only safe semantics.

6 Broadcast

This section shows how to implement two *broadcast primitives* tolerating Byzantine faults with asymmetric quorums. Recall from the standard literature [14, 8, 4] that reliable broadcasts offer basic forms of reliable message delivery and consistency, but they do not impose a total order on delivered messages (as this is equivalent to consensus). The Byzantine broadcast primitives described here, *consistent broadcast* and *reliable broadcast*, are prominent building blocks for many more advanced protocols.

With both primitives, the sender process may broadcast a message m by invoking *broadcast*(m); the broadcast abstraction outputs m to the local application on the process through a *deliver*(m) event. Moreover, the notions of broadcast considered in this section are intended to deliver only one message per instance. Every instance has a distinct (implicit) label and a designated sender p_s. With standard multiplexing techniques one can extend this to a protocol in which all processes may broadcast messages repeatedly [4].

Byzantine consistent broadcast. The simplest such primitive, which has been called *(Byzantine) consistent broadcast* [4], ensures only that those correct processes which deliver a message agree on the content of the message, but they may not agree on termination. In other words, the primitive does not enforce "reliability" such that a correct process outputs a message if and only if all other correct processes produce an output. The events in its interface are denoted by *c-broadcast* and *c-deliver*.

The change of the definition towards asymmetric quorums affects most of its guarantees, which hold only for wise processes but not for all correct ones. This is similar to the definition of a register in Section 5.

▶ **Definition 11** (Asymmetric Byzantine consistent broadcast)**.** *A protocol for* asymmetric (Byzantine) consistent broadcast *satisfies:*

Validity: *If a correct process p_s c-broadcasts a message m, then all wise processes eventually c-deliver m.*

Algorithm 2 Asymmetric Byzantine consistent broadcast protocol with sender p_s (process p_i).

State
  sentecho $\leftarrow$ FALSE: indicates whether p_i has sent ECHO
  echos $\leftarrow [\perp]^N$: collects the received ECHO messages from other processes
  delivered $\leftarrow$ FALSE: indicates whether p_i has delivered a message

upon invocation *c-broadcast*(m) **do**
  send message $[\text{SEND}, m]$ to all $p_j \in \mathcal{P}$

upon receiving a message $[\text{SEND}, m]$ from p_s **such that** $\neg$*sentecho* **do**
  sentecho $\leftarrow$ TRUE
  send message $[\text{ECHO}, m]$ to all $p_j \in \mathcal{P}$

upon receiving a message $[\text{ECHO}, m]$ from p_j **do**
  if *echos*$[j] = \perp$ **then**
    echos$[j] \leftarrow m$

upon exists $m \neq \perp$ **such that** $\{p_j \in \mathcal{P} | echos[j] = m\} \in \mathcal{Q}_i$ **and** $\neg$*delivered* **do**
  delivered $\leftarrow$ TRUE
  output *c-deliver*(m)

Consistency: *If some wise process c-delivers m and another wise process c-delivers m', then $m = m'$.*

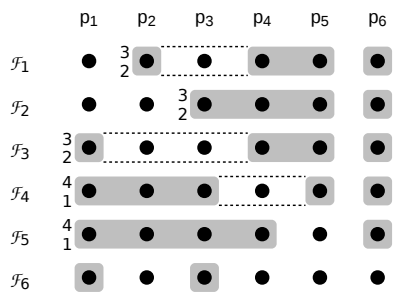
Integrity: *For any message m, every correct process c-delivers m at most once. Moreover, if the sender p_s is correct and the receiver is wise, then m was previously c-broadcast by p_s.*

The following protocol is an extension of "authenticated echo broadcast" [4], which goes back to Srikanth and Toueg [26]. It is a building block found in many Byzantine fault-tolerant protocols with greater complexity. The adaptation for asymmetric quorums is straightforward: Every process considers its own quorums before *c-delivering* the message.

▶ **Theorem 12.** *Algorithm 2 implements asymmetric Byzantine consistent broadcast.*

Example. We illustrate the broadcast protocols using a six-process asymmetric quorum system $\mathbb{Q}_B$, defined through its fail-prone system $\mathbb{F}_B$. In $\mathbb{F}_B$, as shown below, for p_1, p_2, and p_3, each process always trusts itself, some other process of $\{p_1, p_2, p_3\}$ and one further process in $\{p_1, \ldots, p_5\}$. Process p_4 and p_5 each assumes that at most one other process of $\{p_1, \ldots, p_5\}$ may fail (excluding itself). Moreover, none of the processes p_1, …, p_5 ever trusts p_6. For p_6 itself, the fail-prone set is $\{p_1, p_3\}$, i.e., it trusts p_2, p_4, and p_5 unconditionally.

$$\mathbb{F}_B: \quad \begin{aligned} \mathcal{F}_1 &= \Theta^3_2(\{p_2, p_4, p_5\}) * \{\{p_6\}\} \\ \mathcal{F}_2 &= \Theta^3_2(\{p_3, p_4, p_5\}) * \{\{p_6\}\} \\ \mathcal{F}_3 &= \Theta^3_2(\{p_1, p_4, p_5\}) * \{\{p_6\}\} \\ \mathcal{F}_4 &= \Theta^4_1(\{p_1, p_2, p_3, p_5\}) * \{\{p_6\}\} \\ \mathcal{F}_5 &= \Theta^4_1(\{p_1, p_2, p_3, p_4\}) * \{\{p_6\}\} \\ \mathcal{F}_6 &= \{\{p_1, p_3\}\} \end{aligned}$$



One can verify that $B^3(\mathbb{F}_B)$ holds; hence, let $\mathbb{Q}_B$ be the canonical quorum system of $\mathbb{F}_B$. Again, there is no reliable process that could be trusted by all and $\mathbb{Q}_B$ is not a special case of a symmetric threshold Byzantine quorum system. With $F = \{p_1, p_5\}$, for instance, process p_3 is wise, p_2, p_4, and p_6 are naïve, and there is no guild.

Consider now an execution of Algorithm 2 with sender p_4^* and $F = \{p_4^*, p_5^*\}$ (we write p_4^* and p_5^* to denote that they are faulty). This means processes p_1, p_2, p_3 are wise and form a guild because $\{p_1, p_2, p_3\}$ is a quorum for all three; furthermore, p_6 is naïve. The protocol execution proceeds as follows.

$$p_4^* : \begin{cases} [\textsc{send}, x] \to p_1, p_3 \\ [\textsc{send}, u] \to p_2, p_6 \end{cases}$$

$$\begin{array}{lll} p_1 : & [\textsc{echo}, x] \to \mathcal{P} & p_1 : \textit{c-deliver}(x) \\ p_2 : & [\textsc{echo}, u] \to \mathcal{P} & p_2 : \text{no quorum of } [\textsc{echo}] \text{ in } \mathcal{Q}_2 \\ p_3 : & [\textsc{echo}, x] \to \mathcal{P} & p_3 : \text{no quorum of } [\textsc{echo}] \text{ in } \mathcal{Q}_3 \\ & p_4^* : \begin{cases} [\textsc{echo}, x] \to p_1 \\ [\textsc{echo}, u] \to p_6 \end{cases} & \\ & p_5^* : \begin{cases} [\textsc{echo}, x] \to p_1 \\ [\textsc{echo}, u] \to p_6 \end{cases} & \\ p_6 : & [\textsc{echo}, u] \to \mathcal{P} & p_6 : \textit{c-deliver}(u) \end{array}$$

Hence, p_1 receives $[\textsc{echo}, x]$ from, say, $\{p_1, p_3, p_4^*\} \in \mathcal{Q}_1$ and *c-delivers* x, but the other wise processes do not terminate. The naïve p_6 gets $[\textsc{echo}, u]$ from $\{p_2, p_4^*, p_5^*, p_6\} \in \mathcal{Q}_6$ and *c-delivers* $u \neq x$.

Byzantine reliable broadcast. In the symmetric setting, consistent broadcast has been extended to *(Byzantine) reliable broadcast* in a well-known way to address the disagreement about termination among the correct processes [4]. This primitive has the same interface as consistent broadcast, except that its events are called *r-broadcast* and *r-deliver* instead of *c-broadcast* and *c-deliver*, respectively.

A reliable broadcast protocol also has all properties of consistent broadcast, but satisfies the additional *totality* property stated next. Taken together, *consistency* and *totality* imply a notion of *agreement*, similar to what is also ensured by many crash-tolerant broadcast primitives. Analogously to the earlier primitives with asymmetric trust, our notion of an *asymmetric reliable broadcast*, defined next, ensures agreement on termination only for the wise processes, and moreover only for executions with a guild. Also the *validity* of Definition 11 is extended by the assumption of a guild. Intuitively, one needs a guild because the wise processes that make up the guild are self-sufficient, in the sense that the guild contains a quorum of wise processes for each of its members; without that, there may not be enough wise processes.

▶ **Definition 13** (Asymmetric Byzantine reliable broadcast)**.** *A protocol for* asymmetric (Byzantine) reliable broadcast *is a protocol for asymmetric Byzantine consistent broadcast with the revised* validity *condition and the additional* totality *condition stated next:*

Validity: *In all executions with a guild, if a correct process p_s c-broadcasts a message m, then all processes in the maximal guild eventually c-deliver m.*

Totality: *In all executions with a guild, if a wise process r-delivers some message, then all processes in the maximal guild eventually r-deliver a message.*

The protocol of Bracha [3] implements reliable broadcast subject to Byzantine faults with symmetric trust. It augments the authenticated echo broadcast from Algorithm 2 with a second all-to-all exchange, where each process is supposed to send READY with the

payload message that will be *r-delivered*. When a process receives the same m in $2f+1$ READY messages, in the symmetric model with a threshold Byzantine quorum system, then it *r-delivers* m. Also, a process that receives $[\text{READY}, m]$ from $f+1$ distinct processes and that has not yet sent a READY chimes in and also sends $[\text{READY}, m]$. These two steps ensure totality.

For asymmetric quorums, the conditions of a process p_i receiving $f+1$ and $2f+1$ equal READY messages, respectively, generalize to receiving the same message from a kernel for p_i and from a quorum for p_i. Intuitively, the change in the first condition ensures that when a wise process p_i receives the same $[\text{READY}, m]$ message from a kernel for itself, then this kernel intersects with some quorum of wise processes. Therefore, at least one wise process has sent $[\text{READY}, m]$ and p_i can safely adopt m. Furthermore, the change in the second condition relies on the properties of asymmetric quorums to guarantee that whenever some wise process has *r-delivered* m, then enough correct processes have sent a $[\text{READY}, m]$ message such that all wise processes eventually receive a kernel of $[\text{READY}, m]$ messages and also send $[\text{READY}, m]$.

Applying these changes to Bracha's protocol results in the asymmetric reliable broadcast protocol shown in Algorithm 3. Note that it strictly extends Algorithm 2 by the additional round of READY messages, in the same way as for symmetric trust. For instance, when instantiated with the symmetric threshold quorum system of $n = 3f+1$ processes, of which f may fail, then every set of $f+1$ processes is a kernel.

In Algorithm 3, there are two conditions that let a correct p_i send $[\text{READY}, m]$: either receiving a quorum of $[\text{ECHO}, m]$ messages for itself or obtaining a kernel for itself of $[\text{READY}, m]$ messages. For the first case, we say p_i *sends* READY *after* ECHO; for the second case, we say p_i *sends* READY *after* READY.

▶ **Lemma 14.** *In any execution with a guild, there exists a unique m such that whenever a wise process sends a* READY *message, it contains m.*

This lemma follows from the fact that Algorithm 3 extends Algorithm 2 for consistent broadcast, combined with the consistency property in Definition 11. This shows why the lemma holds for READY messages sent by wise processes after ECHO. For READY messages sent after READY, a new argument is needed, which relies on the properties of kernels and appears in the long version [5].

▶ **Theorem 15.** *Algorithm 3 implements asymmetric Byzantine reliable broadcast.*

Example. Consider again the protocol execution with $\mathbb{Q}_B$ introduced earlier for illustrating asymmetric consistent broadcast. Recall that $F = \{p_4^*, p_5^*\}$, the set $\{p_1, p_2, p_3\}$ is a guild, and p_6 is naïve. The start of the execution is the same as shown previously and omitted. Instead of *c-delivering* x and u, respectively, p_1 and p_6 send $[\text{READY}, x]$ and $[\text{READY}, u]$ to all processes:

...	$p_1 : [\text{READY}, x] \to \mathcal{P}$			$p_1 : r\text{-}deliver(x)$
...	$p_2 :$ no quorum	$p_2 : [\text{READY}, x] \to \mathcal{P}$		$p_2 : r\text{-}deliver(x)$
...	$p_3 :$ no quorum		$p_3 : [\text{READY}, x] \to \mathcal{P}$	$p_3 : r\text{-}deliver(x)$
...	$p_4^* : -$			
...	$p_5^* : -$			
...	$p_6 : [\text{READY}, u] \to \mathcal{P}$	$p_6 :$ no quorum		

Algorithm 3 Asymmetric Byzantine reliable broadcast protocol with sender p_s (process p_i).

State
  sentecho ← FALSE: indicates whether p_i has sent ECHO
  echos ← $[\bot]^N$: collects the received ECHO messages from other processes
  sentready ← FALSE: indicates whether p_i has sent READY
  readys ← $[\bot]^N$: collects the received READY messages from other processes
  delivered ← FALSE: indicates whether p_i has delivered a message

upon invocation *r-broadcast*(m) **do**
  send message $[\text{SEND}, m]$ to all $p_j \in \mathcal{P}$

upon receiving a message $[\text{SEND}, m]$ from p_s **such that** $\neg$*sentecho* **do**
  sentecho ← TRUE
  send message $[\text{ECHO}, m]$ to all $p_j \in \mathcal{P}$

upon receiving a message $[\text{ECHO}, m]$ from p_j **do**
  if $echos[j] = \bot$ **then**
    $echos[j] \leftarrow m$

upon exists $m \neq \bot$ **such that** $\{p_j \in \mathcal{P} | echos[j] = m\} \in \mathcal{Q}_i$ **and** $\neg$*sentready* **do**
  // a quorum for p_i
  sentready ← TRUE
  send message $[\text{READY}, m]$ to all $p_j \in \mathcal{P}$

upon exists $m \neq \bot$ **such that** $\{p_j \in \mathcal{P} | readys[j] = m\} \in \mathcal{K}_i$ **and** $\neg$*sentready* **do**
  // a kernel for p_i
  sentready ← TRUE
  send message $[\text{READY}, m]$ to all $p_j \in \mathcal{P}$

upon receiving a message $[\text{READY}, m]$ from p_j **do**
  if $readys[j] = \bot$ **then**
    $readys[j] \leftarrow m$

upon exists $m \neq \bot$ **such that** $\{p_j \in \mathcal{P} | readys[j] = m\} \in \mathcal{Q}_i$ **and** $\neg$*delivered* **do**
  delivered ← TRUE
  output *r-deliver(m)*

Note that the kernel systems of processes p_1, p_2, and p_3 are, respectively, $\mathcal{K}_1 = \{\{p_1\}, \{p_3\}\}$, $\mathcal{K}_2 = \{\{p_1\}, \{p_2\}\}$, and $\mathcal{K}_3 = \{\{p_2\}, \{p_3\}\}$. Hence, when p_2 receives $[\text{READY}, x]$ from p_1, it sends $[\text{READY}, x]$ in turn because $\{p_1\}$ is a kernel for p_2, and when p_3 receives this message, then it sends $[\text{READY}, x]$ because $\{p_2\}$ is a kernel for p_3.

Furthermore, since $\{p_1, p_2, p_3\}$ is the maximal guild and contains a quorum for each of its members, all three wise processes *r-deliver* x as implied by *consistency* and *totality*. The naïve p_6 does not *r-deliver* anything, however.

Remarks. Asymmetric reliable broadcast (Definition 13) ensures validity and totality only for processes in the maximal guild. On the other hand, an asymmetric consistent broadcast (Definition 11) ensures validity also for all *wise* processes. We leave it as an open problem to determine whether these guarantees can also be extended to wise processes for asymmetric reliable broadcast and the Bracha protocol. This question is equivalent to determining whether there exist any wise processes outside the maximal guild.

Another open problem concerns the conditions for reacting to READY messages in the asymmetric reliable broadcast protocol. Already in Bracha's protocol for the threshold model [3], a process (1) sends its own READY message upon receiving $f + 1$ READY messages and (2) *r-delivers* an output upon receiving $2f + 1$ READY messages. These conditions generalize for arbitrary, non-threshold quorum systems to receiving messages (1) from any set that is guaranteed to contain at least one correct process and (2) from any set that still contains at least one process even when any two fail-prone process sets are subtracted. In Algorithm 3, in contrast, a process delivers the payload only after receiving READY messages from one of its quorums. But such a quorum (e.g., $\lceil \frac{n+f+1}{2} \rceil$ processes) may be larger than a set in the second case (e.g., $2f + 1$ processes). It remains interesting to find out whether this discrepancy is necessary.

References

1. 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 Vukolić, Sharon Weed, and Jason Yellick. Hyperledger Fabric: A distributed operating system for permissioned blockchains. In *Proc. 13th European Conference on Computer Systems (EuroSys)*, pages 30:1–30:15, April 2018. doi:10.1145/3190508.3190538.
2. Frederik Armknecht, Ghassan O. Karame, Avikarsha Mandal, Franck Youssef, and Erik Zenner. Ripple: Overview and outlook. In Mauro Conti, Matthias Schunter, and Ioannis G. Askoxylakis, editors, *Proc. Trust and Trustworthy Computing (TRUST)*, volume 9229 of *Lecture Notes in Computer Science*, pages 163–180. Springer, 2015.
3. Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75:130–143, 1987.
4. Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming (Second Edition)*. Springer, 2011.
5. Christian Cachin and Björn Tackmann. Asymmetric distributed trust. e-print, arXiv:1906.09314 [cs.DC], 2019. URL: http://arxiv.org/abs/1906.09314.
6. Christian Cachin and Marko Vukolić. Blockchain consensus protocols in the wild. In Andréa W. Richa, editor, *Proc. 31st Intl. Symposium on Distributed Computing (DISC 2017)*, volume 91 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:16, 2017. doi:10.4230/LIPIcs.DISC.2017.1.
7. Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, 2002.
8. Bernadette Charron-Bost, Fernando Pedone, and André Schiper, editors. *Replication: Theory and Practice*, volume 5959 of *Lecture Notes in Computer Science*. Springer, 2010.
9. Brad Chase and Ethan MacBrough. Analysis of the XRP ledger consensus protocol. e-print, arXiv:1802.07242 [cs.DC], 2018.
10. Ivan Damgård, Yvo Desmedt, Matthias Fitzi, and Jesper Buus Nielsen. Secure protocols with asymmetric trust. In Kaoru Kurosawa, editor, *Advances in Cryptology: ASIACRYPT 2007*, volume 4833 of *Lecture Notes in Computer Science*. Springer, 2007.
11. Hector Garcia-Molina and Daniel Barbara. How to assign votes in a distributed system. *jacm*, 32(4):841–860, 1985.
12. Álvaro García-Pérez and Alexey Gotsman. Federated Byzantine quorum systems. In *Proc. 22nd Conference on Principles of Distributed Systems (OPODIS)*, volume 125 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:16, 2018. doi:10.4230/LIPIcs.OPODIS.2018.17.
13. David K. Gifford. Weighted voting for replicated data. In *Proc. 7th ACM Symposium on Operating System Principles (SOSP)*, pages 150–162, 1979.

14 Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. In Sape J. Mullender, editor, *Distributed Systems (2nd Ed.)*. ACM Press & Addison-Wesley, New York, 1993.

15 Martin Hirt and Ueli Maurer. Player simulation and general adversary structures in perfect multi-party computation. *Journal of Cryptology*, 13(1):31–60, 2000.

16 Flavio P. Junqueira, Keith Marzullo, Maurice Herlihy, and Lucia Draque Penso. Threshold protocols in survivor set systems. *Distributed Computing*, 23:135–149, 2010.

17 Leslie Lamport. On interprocess communication. *Distributed Computing*, 1(2):77–85, 86–101, 1986.

18 Marta Lokhava, Giuliano Losa, David Mazières, Graydon Hoare, Nicolas Barry, Eli Gafni, Jonathan Jove, Rafal Malinowsky, and Jed McCaleb. Fast and secure global payments with stellar. In *Proc. 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 80–96, 2019.

19 Giuliano Losa, Eli Gafni, and David Mazières. Stellar consensus by instantiation. In Jukka Suomela, editor, *Proc. 33rd International Symposium on Distributed Computing (DISC 2019)*, volume 146 of *LIPIcs*, pages 27:1–27:15, 2019. `doi:10.4230/LIPIcs.DISC.2019.27`.

20 Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, 1996.

21 Dahlia Malkhi and Michael K. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

22 David Mazières. The Stellar consensus protocol: A federated model for Internet-level consensus. Stellar, available online, `https://www.stellar.org/papers/stellar-consensus-protocol.pdf`, 2016.

23 Moni Naor and Avishai Wool. The load, capacity and availability of quorum systems. *SIAM Journal on Computing*, 27(2):423–447, 1998.

24 M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

25 David Schwartz, Noah Youngs, and Arthur Britto. The Ripple protocol consensus algorithm. Ripple Labs, available online, `https://ripple.com/files/ripple_consensus_whitepaper.pdf`, 2014.

26 T. K. Srikanth and Sam Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2:80–94, 1987.

Uniform Partition in Population Protocol Model Under Weak Fairness

Hiroto Yasumi
Nara Institute of Science and Technology, Ikoma city, Nara, Japan
yasumi.hiroto.yf9@is.naist.jp

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

Michiko Inoue
Nara Institute of Science and Technology, Ikoma city, Nara, Japan
kounoe@is.naist.jp

Abstract

We focus on a uniform partition problem in a population protocol model. The uniform partition problem aims to divide a population into k groups of the same size, where k is a given positive integer. In the case of $k = 2$ (called uniform bipartition), a previous work clarified space complexity under various assumptions: 1) an initialized base station (BS) or no BS, 2) weak or global fairness, 3) designated or arbitrary initial states of agents, and 4) symmetric or asymmetric protocols, except for the setting that agents execute a protocol from arbitrary initial states under weak fairness in the model with an initialized base station. In this paper, we clarify the space complexity for this remaining setting. In this setting, we prove that P states are necessary and sufficient to realize asymmetric protocols, and that $P + 1$ states are necessary and sufficient to realize symmetric protocols, where P is the known upper bound of the number of agents. From these results and the previous work, we have clarified the solvability of the uniform bipartition for each combination of assumptions. Additionally, we newly consider an assumption on a model of a non-initialized BS and clarify solvability and space complexity in the assumption. Moreover, the results in this paper can be applied to the case that k is an arbitrary integer (called uniform k-partition).

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Theory of computation → Concurrent algorithms

Keywords and phrases population protocol, uniform k-partition, distributed protocol

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.8

Related Version A full version of the paper is available at `http://arxiv.org/abs/1911.04678`

Funding This work was supported in part by JSPS KAKENHI Grant Number 18K11167 and JST SICORP Grant Number JPMJSC1806.

1 Introduction

1.1 The Background

A population protocol model [6, 9] is an abstract model for devices with heavily limited computation and communication capability. The devices are represented as passively moving agents, and a set of agents is called a population. In this model, if two agents approach, an interaction happens between them. At the time of the interaction, the two agents update their states. By repeating such interactions, agents proceed with computation. The population protocol model has many application examples such as sensor networks and molecular robot networks. For example, one may construct a network to investigate the ecosystem by attaching sensors to a flock of wild small animals such as birds. In this system,

© Hiroto Yasumi, Fukuhito Ooshita, and Michiko Inoue;
licensed under Creative Commons License CC-BY
23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller; Article No. 8; pp. 8:1–8:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Table 1 The minimum number of states to solve the uniform bipartition problem under global fairness.

BS	initial states of agents	symmetry	upper bound	lower bound	paper
initialized BS	designated	asymmetric	3	3	[32]
		symmetric	3	3	
	arbitrary	asymmetric	4	4	
		symmetric	4	4	
non-initialized BS	designated	asymmetric	3	3	
		symmetric	3	3	
	arbitrary	asymmetric	unsolvable		this paper
		symmetric	unsolvable		
no BS	designated	asymmetric	3	3	[32]
		symmetric	4	4	
	arbitrary	asymmetric	unsolvable		
		symmetric	unsolvable		

Table 2 The minimum number of states to solve the uniform bipartition problem under weak fairness.

BS	initial states of agents	symmetry	upper bound	lower bound	paper
initialized BS	designated	asymmetric	3	3	[32]
		symmetric	3	3	
	arbitrary	asymmetric	P	P	this paper
		symmetric	$P+1$	$P+1$	
non-initialized BS	designated	asymmetric	3	3	[32]
		symmetric	3	3	
	arbitrary	asymmetric	unsolvable		this paper
		symmetric	unsolvable		
no BS	designated	asymmetric	3	3	[32]
		symmetric	unsolvable		
	arbitrary	asymmetric	unsolvable		
		symmetric	unsolvable		

sensors exchange information with each other when two sensors approach sufficiently close. By repeating such information exchange, the system eventually grasps the entire environment of the flock. Another example is a system of molecular robots [26]. In this system, a large number of robots cooperate in a human body to achieve an objective (e.g. carrying medicine). To realize such systems, various fundamental protocols have been proposed in the population protocol model [11]. For example, there are leader election protocols [5, 14, 15, 17, 23, 28], counting protocols [10, 12, 13], majority protocols [7, 20], naming protocols [16], and so on.

In this paper, we study a uniform k-partition problem, which divides a population into k groups of the same size, where k is a given positive integer. The uniform k-partition problem has some applications. For example, we can save the battery by switching on only some groups. Another example is to execute multiple tasks by assigning different tasks to each group simultaneously. Protocols for the uniform k-partition problem can be used to attain fault-tolerance [18].

As a previous work, Yasumi et al. [32, 33] studied space complexity of uniform partition when the number of partitions is two (called uniform bipartition). In the paper, they considered four types of assumptions: 1) an initialized base station (BS) or no BS, 2) designated or arbitrary initial states of agents, 3) asymmetric or symmetric protocols, and 4)

Table 3 The minimum number of states to solve the uniform k-partition problem.

fairness	BS	initial states of agents	symmetry	upper bound	lower bound
weak fairness	single	arbitrary	asymmetric	P	P
			symmetric	$P+1$	$P+1$
global fairness	no	designated	symmetric	$3k-2$ [30]	k (truism)

global or weak fairness. A BS is a special agent that is distinguishable from other agents and has powerful capability. An initialized BS means that the BS has a designated initial state in the initial configuration. The BS enables us to construct efficient protocols, though it is sometimes difficult to implement. The assumption of initial states bear on the requirement of initialization and the fault-tolerant property. If a protocol requires designated initial states, it is necessary to initialize all agents to execute the protocol. Alternatively, if a protocol solves the problem with arbitrary initial states, we do not need to initialize agents other than the BS. This implies that, when agents transit to arbitrary states by transient faults, the protocol can reach the desired configuration by initializing the BS. Symmetry of protocols is related to the power of symmetry breaking in the population. Asymmetric protocols may include asymmetric transitions that make agents with the same states transit to different states. This needs a mechanism to break symmetry among agents and its implementation is not easy with heavily limited devices. Symmetric protocols do not include such asymmetric transitions. Fairness is an assumption of interaction patterns. Though weak fairness guarantees only that every pair of agents interact infinitely often, global fairness makes a stronger assumption on the order of interactions.

For most combinations of assumptions, Yasumi et al. [32] clarified the solvability of the uniform bipartition problem and the minimum number of states to solve the problem. Tables 1 and 2 show the solvability of the uniform bipartition. These tables show the number of states to solve the uniform bipartition problem under various assumptions, where P is the known upper bound of the number of agents. The remaining case for an initialized BS and no BS is a protocol with an initialized BS and arbitrary initial states under weak fairness. For this case, they proved only that $P-2$ states are necessary. In this paper, we will give tight lower and upper bounds of the number of states for this case. In addition, recently Burman et al. [16] have considered the case with a non-initialized BS, which is distinguished from other agents but has an arbitrary initial state, for a naming problem. Because Yasumi et al. [32] did not consider the case, we also consider the case in this paper.

For the general case of an arbitrary number of partitions, Yasumi et al. [32] proposed a symmetric protocol with no BS and designated initial sates under global fairness. The protocol uses $3k-2$ states for an agent to construct k groups of the same size. However, no protocol has been proposed for other combinations of assumptions.

1.2 Our Contributions

Our main contribution is to clarify the solvability of the uniform bipartition problem with arbitrary initial states under weak fairness in the model with an initialized BS. A previous work [32] proved only that $P-2$ states are necessary for each agent, where P is the known upper bound of the number of agents. In this paper, we improve the lower bound from $P-2$ states to P states for asymmetric protocols and from $P-2$ states to $P+1$ states for symmetric protocols. Additionally, we propose an asymmetric protocol with P states, and obtain a symmetric protocol with $P+1$ states by a scheme proposed in [12].

Another contribution is to clarify the solvability in case of a non-initialized BS for the uniform bipartition problem. For designated initial states, the protocol with an initialized BS, which is proposed in [32] can still work even if the BS is non-initialized. In this paper, we prove the impossibility with arbitrary initial states in case of non-initialized BS.

By combining these results with the previous work [32], we have clarified the tight upper and lower bounds on the number of states for an agent to solve the uniform bipartition problem for all combinations of assumptions (see Tables 1 and 2).

For the case of an initialized BS, arbitrary initial states, and weak fairness, it is interesting to compare these results with those of naming protocols [16]. A naming protocol aims to assign different states to all agents, and hence it can be regarded as a uniform P-partition protocol (the size of each group is zero or one). Burman et al. [16] prove that, to realize naming protocols in the same setting, P states are necessary and sufficient for asymmetric protocols and $P + 1$ states are necessary and sufficient for symmetric protocols. That is, naming protocols have the same space complexity as uniform k-partition protocols. Clearly naming protocols (or uniform P-partition protocols) require P states to assign different states to P agents. Interestingly uniform bipartition protocols still require P states in this setting. On the other hand, the number of states is reduced to three or four when we assume designated initial states or global fairness.

Protocols proposed in this paper are available for the uniform k-partition problem, where k is a given integer. That is, P states and $P+1$ states are sufficient to realize asymmetric and symmetric protocols, respectively, to solve the uniform k-partition problem from arbitrary initial states under weak fairness in the model with an initialized BS. Since the uniform bipartition is a special case of the uniform k-partition, the lower bound for the uniform bipartition problem can be applied to the uniform k-partition problem. That is, P states and $P + 1$ states are necessary to realize asymmetric and symmetric protocols, respectively, under the assumption. Therefore, we have clarified the tight upper and lower bounds of the number of states for the uniform k-partition problem under the assumption (see Table 3).

Due to space constraints, we have omitted some proofs. See [31] for the full version of this paper.

1.3 Related Works

The population protocol model was first introduced in [6, 8]. In those papers, the class of computable predicates in this model was studied. After that, many fundamental tasks have been studied such as leader election, counting, and majority. Those problems have been studied under various assumptions such as existence of a base station, fairness, symmetry of protocols, and initial states of agents. Many researchers have considered the leader election problem for both designated and arbitrary initial states. For designated initial states, leader election protocols have been studied intensively to minimize the time and space complexity [1, 3, 14, 15, 19, 21, 22, 27]. Alistarh et al. [3] proposed an algorithm that solves the problem in polylogarithmic stabilization time with polylogarithmic states. In [19], it was clarified that $\Omega(n)$ parallel time is necessary (i.e., $\Omega(n^2)$ interactions are necessary) to solve the problem with probability 1. After that, many researchers focused on solving the problem with high probability and shrink the time and space complexity [14, 15, 21, 22, 27]. On the other hand, for arbitrary initial states, self-stabilizing and loosely-stabilizing protocols are proposed [9, 17, 23, 28]. The counting problem, which aims to count the number of agents in the population, was introduced by [13]. After that, some researchers have studied the protocol to minimize the space complexity of the counting protocols [12, 24]. In [10], a time and space optimal protocol was proposed. The majority problem is also a fundamental problem that

aims to decide majority of initial states. For the majority problem, many protocols have been studied [1, 2, 4, 7, 15, 20]. Although there are some difference in the model (existence of failure, deterministic or probabilistic solution, and so on), these works also aim to minimize the time and space complexity. Moreover, in recent years, Burman et al. [16] proposed a naming protocol which assigns a different state (called name) to each agent. In the paper, they completely clarify the solvability of the naming protocol under various assumptions.

The uniform k-partition problem and a similar problem have been considered in [16, 25, 29, 30]. Lamani et al. [25] studied a group composition problem, which aims to divide a population into groups of designated sizes. They assume that half of agents make interactions at the same time and that all agents know n. Therefore the protocol does not work in our setting. In [30], Yasumi et al. proposed a uniform k-partition protocol that requires $3k-2$ states without the BS under global fairness. Moreover, some of the authors extended the result of [30] to the R-generalized partition problem, where the protocol divides all agents into k groups whose sizes follow a given ratio R [29]. Since they assume designated initial states and global fairness, the protocol does not work in our setting. In addition, Delporte-Gallet et al. [18] proposed a protocol solving the k-partition problem with less uniformity. This protocol guarantees that each group includes at least $n/(2k)$ agents, where n is the number of agents. This protocol requires $k(k+3)/2$ states under global fairness.

2 Definitions

2.1 Population Protocol Model

A population is a collection of pairwise interacting agents, denoted by A. A protocol $\mathcal{P}(Q,\delta)$ consists of Q and δ, where Q is a set of possible states of agents and δ is a set of transitions on Q. Each transition in δ is denoted by $(p,q) \to (p',q')$, which means that, when an agent with state p and an agent with state q interact, they transit their states to p' and q', respectively. Transition $(p,q) \to (p',q')$ is null if both $p=p'$ and $q=q'$ hold. We omit null transitions in descriptions of algorithms. Transition $(p,q) \to (p',q')$ is asymmetric if both $p=q$ and $p' \neq q'$ hold; otherwise, the transition is symmetric. Protocol $\mathcal{P}(Q,\delta)$ is symmetric if every transition in δ is symmetric. Protocol $\mathcal{P}(Q,\delta)$ is asymmetric if every transition in δ is symmetric or asymmetric. Protocol $\mathcal{P}(Q,\delta)$ is deterministic if, for any pair of states $(p,q) \in Q \times Q$, exactly one transition $(p,q) \to (p',q')$ exists in δ. We consider only deterministic protocols in this paper. A global state of a population is called a configuration, defined as a vector of (local) states of all agents. A state of agent a in configuration C, is denoted by $s(a,C)$. Moreover, when C is clear from the context, we simply denote $s(a)$. Transition of configurations is described in the form $C \to C'$, which means that configuration C' is obtained from C by a single transition of a pair of agents. For configurations C and C', if there exists a sequence of configurations $C = C_0, C_1, \ldots, C_m = C'$ such that $C_i \to C_{i+1}$ holds for any i $(0 \leq i < m)$, we say C' is reachable from C, denoted by $C \xrightarrow{*} C'$. An infinite sequence of configurations $E = C_0, C_1, C_2, \ldots$ is an execution of a protocol if $C_i \to C_{i+1}$ holds for any i $(i \geq 0)$. An execution E is weakly-fair if every pair of agents a and a' interacts infinitely often. An execution segment is a subsequence of an execution.

In this paper, we assume that a single BS exists in A. The BS is distinguishable from other non-BS agents, although non-BS agents cannot be distinguished. That is, state set Q is divided into state set Q_b of a BS and state set Q_p of non-BS agents. The BS has unlimited resources, in contrast with resource-limited non-BS agents. That is, we focus on the number of states $|Q_p|$ for non-BS agents and do not care the number of states $|Q_b|$ for the BS. For this reason, we say a protocol uses x states if $|Q_p| = x$ holds. Throughout the paper, we

assume that non-BS agents have arbitrary initial states. On the other hand, as for the BS, we consider two cases, an initialized BS and a non-initialized BS. When we assume an initialized BS, the BS has a designated initial state while all non-BS agents have arbitrary initial states. When we assume a non-initialized BS, the BS also has an arbitrary initial state. For simplicity, we use agents only to refer to non-BS agents in the following sections. To refer to the BS, we always use the BS (not an agent). In the initial configuration, the BS and non-BS agents do not know the number of agents, but they know the upper bound P of the number of agents.

2.2 Uniform k-Partition Problem

Let A_p be a set of all non-BS agents. Let $f : Q_p \to \{color_1, color_2, \ldots, color_k\}$ be a function that maps a state of a non-BS agent to $color_i (1 \leq i \leq k)$. We define a color of $a \in A_p$ as $f(s(a))$. We say agent $a \in A_p$ belongs to the i-th group if $f(s(a)) = color_i$ holds.

Configuration C is stable if there is a partition $\{G_1, G_2, \ldots, G_k\}$ of A_p that satisfies the following condition:

1. $||G_i| - |G_j|| \leq 1$ for any i and j, and
2. For all C^* such that $C \xrightarrow{*} C^*$, each agent in G_i belongs to the i-th group at C^* (i.e., at C^*, any agent a in G_i satisfies $f(s(a)) = color_i$).

An execution $E = C_0, C_1, C_2, \ldots$ solves the uniform k-partition problem if E includes a stable configuration C_t. If every weakly-fair execution E of protocol $\mathcal{P}$ solves the uniform k-partition problem, we say protocol $\mathcal{P}$ solves the uniform k-partition problem under weak fairness.

3 Impossibility Results for Initialized BS and Weak Fairness

In this section, we give impossibility results of asymmetric and symmetric protocols for the uniform bipartition problem (i.e., $k = 2$). Clearly these impossibility results can be applied to the uniform k-partition problem for $k > 2$. Recall that, for an initialized BS, we assume weak fairness and arbitrary initial states.

Since we consider the case of $k = 2$, function f is defined as $f : Q_p \to \{color_1, color_2\}$. In this section, we assign colors *red* and *blue* to $color_1$ and $color_2$, respectively, and we define f as function $f : Q_p \to \{red, blue\}$ that maps a state of a non-BS agent to *red* or *blue*. We say agent $a \in A_p$ is *red* (resp., *blue*) if $f(s(a)) = red$ (resp., $f(s(a)) = blue$) holds. We say s is a c-state if $f(s) = c$ holds. For $c \in \{red, blue\}$, we define c-agent as an agent that has a c-state. We define $\overline{red} = blue$ and $\overline{blue} = red$.

3.1 Common Properties of Asymmetric and Symmetric Protocols

First, we show basic properties that hold for both asymmetric and symmetric protocols. Let *Alg* be a protocol that solves the uniform bipartition. Recall that P is the known upper bound of the number of agents. Hence, *Alg* must solve the uniform bipartition when the actual number of agents is at most P. In the remainder of this subsection, we consider the case that the actual number of agents is $P - 2$.

Lemma 1 shows that, in any execution for $P - 2$ agents, eventually all agents continue to keep different states.

▶ **Lemma 1.** *In any weakly-fair execution $E = C_0, C_1, C_2, \ldots$ of Alg with $P-2$ agents and an initialized BS, there exists a configuration C_h such that 1) C_h is a stable configuration, and, 2) all agents have different states at $C_{h'}$ for any $h' \geq h$.*

Proof. (*Sketch*) For contradiction, we assume that there exist two agents with the same state s in a stable configuration of some execution E with $P-2$ agents. Next, consider an execution with P agents such that two additional agents have s as their initial states and other agents behave similarly to E. In the execution, two additional agents do not join the interactions until $P-2$ agents converge to a stable configuration in E. At that time, two of the $P-2$ agents have state s and additional two agents also have state s. We can prove that, from this configuration, $P-2$ agents cannot recognize the two additional agents and hence they make the same behavior as in E. In addition, the two additional agents can keep state s. Since the numbers of *red* and *blue* agents are balanced without the two additional agents and the two additional agents have the same state, the uniform bipartition problem cannot be solved. This is a contradiction. ◀

In the next lemma, we prove that there exists a configuration C such that, in any configuration reachable from C, all agents have different states. In addition, we also show that the system reaches C in some execution.

▶ **Definition 2.** *Configuration C is strongly-stable if 1) C is stable, and, 2) for any configuration C' with $C \xrightarrow{*} C'$, all agents have different states at C'.*

▶ **Lemma 3.** *When the number of agents other than the BS is $P-2$, there exists an execution of Alg that includes a strongly-stable configuration.*

Proof. (*Sketch*) For contradiction, we assume that such execution does not exist. First, consider a weakly-fair execution E of *Alg*. By Lemma 1, after some configuration C_t in E, all agents have different states. From the assumption, C_t is not strongly-stable. That is, there exists a configuration C_u reachable from C_t such that two agents have the same state. Hence, we can construct another weakly-fair execution E' of *Alg* such that E' is similar to E until C_t and C_u occurs after that. By Lemma 1, after some configuration $C_{t'}$ in E', all agents have different states. Observe that $C_{t'}$ occurs after C_u. From the assumption, there exists a configuration $C_{u'}$ reachable from $C_{t'}$ such that two agents have the same state. Hence, similarly to E', we can construct another weakly-fair execution E'' of *Alg* such that E'' is similar to E' until $C_{t'}$ and $C_{u'}$ occurs after that. By repeating this construction, we can construct a weakly-fair execution such that two agents have the same state infinitely often. From Lemma 1, this is a contradiction. ◀

3.2 Impossibility of Asymmetric Protocols

Here we show the impossibility of asymmetric protocols with $P-1$ states.

▶ **Theorem 4.** *In the model with an initialized BS, there is no asymmetric protocol that solves the uniform bipartition problem with $P-1$ states from arbitrary initial states under weak fairness, if P is an even integer.*

To prove the theorem by contradiction, we assume that such protocol Alg_{asym} exists. Let $Q_p = \{s_1, s_2, \ldots, s_{P-1}\}$ be a state set of agents other than the BS. Let $Q_{blue} = \{s \in Q_p \mid f(s) = blue\}$ be a set of blue states and $Q_{red} = \{s \in Q_p \mid f(s) = red\}$ be a set of red states. Without loss of generality, we assume that $|Q_{blue}| < |Q_{red}|$ holds. Recall that Lemmas 1 and 3 can be applied to both symmetric and asymmetric algorithms. Hence, the properties of

the lemmas hold even in Alg_{asym}. In this proof, based on the properties, we construct an execution of P agents such that the BS does not recognize the difference from the execution of $P-2$ agents. We show contradiction by proving that this execution does not achieve uniform bipartition.

By Lemma 1, clearly Alg_{asym} requires $P/2-1$ *blue* states and $P/2-1$ *red* states. Consequently, we have the following two corollaries.

▶ **Corollary 5.** *$|Q_{blue}| = P/2-1$ and $|Q_{red}| = P/2$ hold.*

▶ **Corollary 6.** *For any weakly-fair execution of Alg_{asym} with $P-2$ agents and an initialized BS, any strongly-stable configuration includes all states in Q_{blue}.*

To prove the main theorem, we focus on the following weakly-fair execution of Alg_{asym} with $P-2$ agents.

▶ **Definition 7.** *Consider a population $A = \{a_0, a_1, \ldots, a_{P-2}\}$ of $P-2$ agents and an initialized BS, where a_0 is the BS. We define $E_\alpha = C_0, C_1, C_2, \ldots$ as a weakly-fair execution of Alg_{asym} for population A that satisfies the following conditions.*

- *E_α includes a strongly-stable configuration C_t, and,*
- *for any $u \geq 0$, agents that interact at $C_{t+2u} \to C_{t+2u+1}$ also interact at $C_{t+2u+1} \to C_{t+2(u+1)}$.*

Note that, in E_α, the system reaches a strongly-stable configuration C_t (this is possible from Lemma 3), and after C_t agents always repeat the same interaction twice.

▶ **Definition 8.** *We define Q_t as a set of states that appear after C_t in E_α.*

Note that, since C_t is strongly-stable, Q_t includes at least $P-2$ states. This implies that Q_t includes all states in Q_p or does not include one state in Q_p. From Corollary 6, $Q_{blue} \subset Q_t$ holds.

The following lemmas give key properties of Alg_{asym} to prove Theorem 4. We will present proofs of these lemmas later.

▶ **Lemma 9.** *For any distinct states p and q ($p \neq q$) such that $p \in Q_{blue}$ and $q \in Q_t$ hold, transition rule $(p,q) \to (p',q')$ satisfies the following conditions.*

- *If $q \in Q_{red}$ or $q \in Q_b$ (i.e., q is a state of the BS) holds, $p' = p$ holds.*
- *If $q \in Q_{blue}$ holds, either $(p',q') = (p,q)$ or $(p',q') = (q,p)$ holds.*

▶ **Lemma 10.** *There is a non-empty state set $Q^* \subseteq Q_{blue}$ that satisfies the following conditions.*

- *For any state $p \in Q^*$, transition rule $(p,p) \to (p',q')$ satisfies $p' \in Q^*$ and $q' \in Q^*$.*
- *Assume that, in a configuration C, there exists a subset of agents A^* such that all agents in A^* have states in Q^* and $|A^*| = |Q^*|+1$ holds. In this case, for any agent $a_r \in A^*$ and any state $q \in Q^*$, there exists an execution segment such that 1) the execution segment starts from C, 2) a_r has state q at the last configuration, 3) only agents in A^* join interactions, and 4) all agents in A^* have states in Q^* at the last configuration.*

Lemma 10 means that, if $|Q^*|+1$ agents have states in Q^*, we can make an arbitrary agent with a state in Q^* transit to an arbitrary state in Q^*. Using these lemmas, we show the theorem by constructing a weakly-fair execution of Alg_{asym} with P agents that cannot be distinguished from execution E_α.

Proof of Theorem 4

Consider a population $A' = \{a'_0, \ldots, a'_P\}$ of P agents and an initialized BS, where a'_0 is the BS. Let C'_0 be an initial configuration such that initial states of $a'_0, \ldots, a'_P$ are $s(a_0, C_0), \ldots, s(a_{P-2}, C_0)$, s^*, s^*, where s^* is a state in Q^*.

For A' we construct an execution $E_\beta = C'_0, C'_1, \ldots, C'_t, \ldots$ using execution E_α as follows.

- For $0 \le u \le t-1$, when a_i and a_j interact at $C_u \to C_{u+1}$ in E_α, a'_i and a'_j interact at $C'_u \to C'_{u+1}$ in E_β.

Clearly, $s(a'_i, C'_t) = s(a_i, C_t)$ holds for any i $(0 \le i \le P-2)$. Since $s(a'_{P-1}, C'_t) = s(a'_P, C'_t) = s^*$ holds, the difference in the numbers of *red* and *blue* agents remains two and consequently C'_t is not a stable configuration.

To construct the remainder of E_β, first let us consider the characteristics of C'_t. Let $A_q \subseteq A$ be a set of agents that have states in Q^* at C_t, and let $\bar{A}_q = A - A_q$. Since all agents have different states and all states in Q_{blue} appear in C_t by Corollary 6, we have $|A_q| = |Q^*|$ from $Q^* \subseteq Q_{blue}$. Let $A'_q \subseteq A'$ be a set of agents that have states in Q^* at C'_t, and let $\bar{A}'_q = A' - A'_q$. Note that, for $i \le P-2$, $a_i \in A_q$ holds if and only if $a'_i \in A'_q$ holds. Since a'_{P-1} and a'_P are also in A'_q, we have $|A'_q| = |Q^*| + 2$. In the following, we construct the remainder of execution E_β that includes infinitely many configurations similar to E_α. we define similarity of configurations in E_β and E_α as follows.

▶ **Definition 11.** *We say configuration C'_u $(u \ge t)$ in E_β is similar to C_v $(v \ge t)$ in E_α if the following conditions hold:*

- *For any agent $a_i \in A_q$, $s(a_i, C_v) \in Q^*$ holds.*
- *For any agent $a'_i \in A'_q$, $s(a'_i, C'_u) \in Q^*$ holds.*
- *For any agent $a'_i \in \bar{A}'_q$ (i.e., $a_i \in \bar{A}_q$), $s(a'_i, C'_u) = s(a_i, C_v)$ holds.*

Let us focus on an execution segment $e = C_{t+2u}, C_{t+2u+1}, C_{t+2(u+1)}$ of E_α for any $u \ge 0$, and consider a configuration C'_x of A' such that C'_x is similar to C_{t+2u}. From now, we explain the way to construct an execution segment $e' = C'_x, \ldots, C'_y$ of E_β that guarantees that C'_y is similar to $C_{t+2(u+1)}$. Since C'_t is similar to C_t, we can repeatedly apply this construction and construct an infinite execution E_β. As a result, for any $u \ge 0$, E_β includes a configuration C' that is similar to C_{t+2u}. Since C' includes $P-1$ *red* agents and $P+1$ *blue* agents, E_β does not include a stable configuration. Note that E_β is not necessarily weakly-fair, but later we explain the way to construct a weakly-fair execution from E_β.

Consider configuration C'_x that is similar to C_{t+2u}. Assume that, in E_α, agents a_i and a_j interact in $C_{t+2u} \to C_{t+2u+1}$. Recall that a_i and a_j also interact in $C_{t+2u+1} \to C_{t+2(u+1)}$. We construct execution segment e' as follows:

- Case that $a_i \in A_q \wedge a_j \in A_q$ holds. Since $s(a_i, C_{t+2u}) \in Q^* \subseteq Q_{blue}$ and $s(a_j, C_{t+2u}) \in Q^* \subseteq Q_{blue}$ hold, $s(a_i, C_{t+2(u+1)}) \in Q^*$ and $s(a_j, C_{t+2(u+1)}) \in Q^*$ also hold from Lemma 10 (the first condition) and Lemma 9. Since other agents do not change their states, C'_x is similar to $C_{t+2(u+1)}$. Hence, in this case, we consider that the constructed execution segment e' is empty.
- Case that either $a_i \in A_q \wedge a_j \in \bar{A}_q$ or $a_i \in \bar{A}_q \wedge a_j \in A_q$ holds. Without loss of generality, we assume that $a_i \in A_q \wedge a_j \in \bar{A}_q$ holds. In this case, $s(a'_i, C'_x) \in Q^*$ is not necessarily equal to $\alpha = s(a_i, C_{t+2u}) \in Q^*$. Hence, in the execution segment e', we first make some agent $a'_r \in A'_q$ enter state α by interactions among agents in A'_q. By Lemma 10 (the second condition) and $|A'_q| = |Q^*| + 2$, such interactions exist and all agents in A'_q have states in Q^* after the interactions. Let C'_z be the resultant configuration.

Clearly C'_z is similar to C_{t+2u} and $s(a'_r, C'_z) = s(a_i, C_{t+2u}) \wedge s(a'_j, C'_z) = s(a_j, C_{t+2u})$ holds. After that, a'_r and a'_j interact twice. We regard the resultant configuration as C'_y (i.e., the last configuration of the constructed execution segment e'). Clearly both $s(a'_r, C'_y) = s(a_i, C_{t+2(u+1)})$ and $s(a'_j, C'_y) = s(a_j, C_{t+2(u+1)})$ hold. Since C'_z is similar to C_{t+2u} and $s(a'_j, C'_y) = s(a_j, C_{t+2(u+1)})$, it is sufficient to prove $s(a'_r, C'_y) \in Q^*$ to guarantee that C'_y is similar to $C_{t+2(u+1)}$. Observe that $s(a_j, C_{t+2u}) \notin Q^*$. This is because, since C_{t+2u} is strongly-stable, all agents have different states and agents in A_q occupy all states in Q^* (the first condition of Definition 11). Hence, $s(a'_r, C'_z) = s(a_i, C_{t+2u}) \in Q^*$ is not equal to $s(a'_j, C'_z) = s(a_j, C_{t+2u}) \notin Q^*$. Consequently, from $s(a'_r, C'_z) \in Q^* \subseteq Q_{blue}$, $s(a'_r, C'_y) = s(a'_r, C'_z) \in Q^*$ holds by Lemma 9. Therefore, C'_y is similar to $C_{t+2(u+1)}$.

- Case that $a_i \in \bar{A}_q \wedge a_j \in \bar{A}_q$ holds. In this case, since $s(a'_i, C'_x) = s(a_i, C_{t+2u})$ and $s(a'_j, C'_x) = s(a_j, C_{t+2u})$ hold, a'_i and a'_j simply interact twice consecutively. We regard the resultant configuration as C'_y (i.e., the last configuration of the constructed execution segment e'). Clearly, since a'_i and a'_j change their states similarly to a_i and a_j, C'_y is similar to $C_{t+2(u+1)}$.

Now we have constructed infinite execution E_β, but E_β is not necessarily weakly-fair. In the following, we construct a weakly-fair execution E_γ of population A' by slightly modifying E_β. To guarantee that E_γ is weakly-fair, for any pair of agents (a'_i, a'_j), a'_i and a'_j should interact infinite number of times in E_γ. For pair of agents (a'_i, a'_j) with $a'_i \in \bar{A}'_q$ and $a'_j \in \bar{A}'_q$, a'_i and a'_j interact infinite number of times in E_β because E_α is weakly-fair and a'_i interacts with a'_j in E_β when a_i interacts with a_j in E_α. For pair of agents (a'_i, a'_j) with $a'_i \in A'_q$ and $a'_j \in A'_q$, we can arbitrarily add interactions of them because, by Lemma 10 (the first condition) and Lemma 9 (the second condition), they keep their states in Q^* and consequently do not influence similarity of configurations.

Hence, we consider the remaining pair (a'_i, a'_j), that is, either $a'_i \in A'_q \wedge a'_j \in \bar{A}'_q$ or $a'_i \in \bar{A}'_q \wedge a'_j \in A'_q$ holds. Without loss of generality, we assume that a'_i is in A'_q and a'_j is in $\bar{A}'_q$. Since E_α is weakly-fair, a_j interacts with an agent in A_q infinite number of times in E_α. Recall that these interactions correspond to interactions of a'_j and a'_r in E_β, and a'_r can be arbitrarily selected from A'_q. For this reason, we can choose a'_r in a round-robin manner so that a'_j interacts with any agent in A'_q infinite number of times. For example, when a_j and an agent in A_q first interact (after C_t), we choose an agent in A'_q as a'_r, and then in the next interaction of a_j and an agent in A_q we can choose another agent in A'_q as a'_r. By this construction, a'_j can interact with any agent a'_i in A'_q infinite number of times.

From this way, we can construct a weakly-fair execution E_γ similarly to E_β. However, for any $u \geq 0$, E_γ includes a configuration C'' that is similar to C_{t+2u}. Since C'' includes $P-1$ *red* agents and $P+1$ *blue* agents, E_γ does not include a stable configuration. This is a contradiction.

The Proof Sketch of Lemma 9

Consider the case that transition $(p, q) \to (p', q')$ occurs at a strongly-stable configuration with $P-2$ agents. By Corollary 6, since any strongly-stable configuration includes all states in Q_{blue}, $(p, q) \to (p', q')$ can occur at the configuration.

First, consider the case that $q \in Q_{red}$ or $q \in Q_b$ holds. For contradiction, assume that $p' \neq p$ holds. By Corollary 6, since an agent with p' exists in the strongly-stable configuration, two agents with p' exist after transition $(p, q) \to (p', q')$. By the definition of strongly-stable configuration, this is a contradiction.

Next, consider the case that $q \in Q_{blue}$ holds. For contradiction, assume that $(p', q') \neq (p, q)$ and $(p', q') \neq (q, p)$ hold. By the definition of stable configuration, p' and q' are *blue*.

Hence, by Corollary 6, since an agent with any state in Q_{blue} exists in the strongly-stable configuration, two agents with the same state in Q_{blue} exist after transition $(p, q) \to (p', q')$. By the definition of strongly-stable configuration, this is a contradiction.

The Proof Sketch of Lemma 10

First, to show the proof sketch, we give some definitions.

▶ **Definition 12.** *For states q and q', we say $q \rightsquigarrow q'$ if there exists a sequence of states $q = q_0, q_1, \cdots, q_k = q'$ such that, for any $i (0 \leq i < k)$, transition rule $(q_i, q_i) \to (q_{i+1}, x_i)$ or $(q_i, q_i) \to (x_i, q_{i+1})$ exists for some x_i.*

▶ **Definition 13.** *For states q and q', we say $q \overset{*}{\rightsquigarrow} q'$ if $x \rightsquigarrow q'$ holds for any x such that $q \rightsquigarrow x$ holds.*

Note that, in these definitions, we consider only interactions of agents with the same state. We say two agents are homonyms if they have the same state. Intuitively, $q \rightsquigarrow q'$ means that an agent with state q can transit to q' by only interactions with homonyms. Also, $q \overset{*}{\rightsquigarrow} q'$ means that, even if an agent with state q transits to any state x by interactions with homonyms, it can still transit from x to q' by interactions with homonyms.

Let $Q_{p^*} = \{q \mid p^* \overset{*}{\rightsquigarrow} q\}$. In this proof, we show that Q_{p^*} satisfies the conditions of Q^* of Lemma 10. Clearly, if homonyms with states in Q_{p*} interact, they transit to states in Q_{p*}. This implies that Q_{p^*} satisfies the first condition of Q^* of the lemma. To prove the second condition, we first show that, when $|Q_{p*}|$ agents have states in Q_{p*} initially, for any $s \in Q_{p*}$, there exists an execution such that only homonyms in the $|Q_{p*}|$ agents interact and eventually some agent transits to state s. To show this, we define a potential function $\Phi(C, s)$ for configuration C and state $s \in Q_{p*}$. Intuitively, $\Phi(C, s)$ represents how far configuration C is from a configuration that includes an agent with state s. To define $\Phi(C, s)$, we define $DtQ(s_i, s)$ as follows.

▶ **Definition 14.** *$DtQ(s_i, s)$ is a function that satisfies the following property.*

- *If $s_i = s$ holds, $DtQ(s_i, s) = 0$ holds.*
- *If $s_i \neq s$ and $s_i \in Q_{p*}$ holds, $DtQ(s_i, s) = \min\{DtQ(s_j^1, s), DtQ(s_j^2, s)\} + 1$ holds when transition rule $(s_i, s_i) \to (s_j^1, s_j^2)$ exists.*
- *If $s_i \notin Q_{p^*}$ holds, $DtQ(s_i, s) = \infty$ holds.*

Intuitively, $DtQ(s_i, s)$ gives the minimum number of interactions to transit from state s_i to state s when allowing only interactions with homonyms. Note that, for any $s_i \in Q_{p*}$, s_i can transit to s because $s_i \rightsquigarrow p \rightsquigarrow s$ holds.

▶ **Definition 15.** *Consider configuration C such that $z = |Q_{p^*}|$ agents $a_1, \ldots, a_z$ have states in Q_{p^*}. In this case, we define potential function $\Phi(C, s)$ as a multi set $\{DtQ(s(a_1, C), s), DtQ(s(a_2, C), s), DtQ(s(a_3, C), s), \ldots, DtQ(s(a_z, C), s)\}$.*

▶ **Definition 16.** *For distinct $\Phi(C_1, s)$ and $\Phi(C_2, s)$, we define a comparative operator of them as follows: Let i be the minimum integer such that the number of i-elements is different in $\Phi(C_1, s)$ and $\Phi(C_2, s)$. If the number of i-elements in $\Phi(C_1, s)$ is smaller than $\Phi(C_2, s)$, we say $\Phi(C_1, s) < \Phi(C_2, s)$.*

From now, we show that there exists an execution such that some agent transits to s. Let C be a configuration with $|Q_{p^*}|$ agents such that all agents have states in Q_{p*} and there does not exist an agent with s in C. Since $|Q_{p^*}|$ agents have states in Q_{p*} in C and there does not

exist an agent with s in C, there exist homonyms in C. When homonyms with a state in Q_{p*} interact, they transit to states in Q_{p*}. These imply that, when homonyms interact at $C \to C'$, either an agent with s or homonyms with a state in Q_{p*} exist in C'. Thus, for contradiction, assume that there exists an infinite execution segment $e = C_0, C_1, C_2, \ldots$ with $|Q_{p^*}|$ agents such that only homonyms interact and any agent never has s in e, where C_0 is a configuration such that all agents have states in Q_{p*}. For e, $\Phi(C_0, s) > \Phi(C_1, s) > \Phi(C_2, s) > \cdots$ holds. This is because, since any $p \in Q_{p^*}$ satisfies $p \rightsquigarrow p^* \rightsquigarrow s$, $DtQ(s(a, C_i), s) > DtQ(s(a, C_{i+1}), s)$ holds for at least one agent a that interacts at $C_i \to C_{i+1}$. Hence, eventually some agent has s in e. By the definition of e, this is a contradiction.

From now, we prove the second condition of Lemma 10. Let A^* be a set of agents such that $|A^*| = |Q_{p^*}| + 1$, and assume that all agents in A^* have states in Q_{p^*}. The existence of the above execution implies that, for any agent $a_r \in A^*$, we can make some agent in $A^* - \{a_r\}$ transits to state $s(a_r)$ by interactions among $A^* - \{a_r\}$. Then, we can make an interaction with homonyms between a_r and an agent with $s(a_r)$. After that, since a_r has a state in Q_{p^*}, all agents in A^* keep states in Q_{p^*}. Hence, in the same way, by making interaction repeatedly between a_r and an agent with $s(a_r)$, a_r can transit to any $q \in Q_{p^*}$ because any $p \in Q_{p^*}$ satisfies $p \rightsquigarrow p^* \rightsquigarrow q$. Therefore, Q_{p^*} satisfies the second condition and thus the lemma holds.

3.3 Impossibility of Symmetric Protocols

In this subsection, we show the impossibility of symmetric protocols with P states. To prove this impossibility, we use ideas of the impossibility proof for the naming protocol [16]. This work shows that, in the model with an initialized BS, there is no symmetric naming protocol with P states from arbitrary initial states under weak fairness. We apply the proof of [16] to the uniform bipartition but, since the treated problem is different, we need to make a non-trivial modification.

▶ **Theorem 17.** *In the model with an initialized BS, there is no symmetric protocol that solves the uniform bipartition problem with P states from arbitrary initial states under weak fairness, if P is an even integer.*

In the case of naming protocols [16], the impossibility proof proves that a unique state (called sink state) always exists. However, in the case of uniform bipartition protocols, sometimes no sink state exists. To treat this situation, we additionally define a sink pair, which is a pair of two states that has a similar property of a sink state. We show that either a sink state or a sink pair exists, and, we prove that there is no symmetric protocol in both cases.

4 Possibility Results for Initialized BS and Weak Fairness

In this section, we propose both asymmetric and symmetric protocols for the uniform k-partition problem. The asymmetric protocol requires P states and the symmetric protocol requires $P + 1$ states. By impossibility results, these protocols are space-optimal.

4.1 An Asymmetric Protocol

In this subsection, we show a P-state asymmetric protocol for the uniform k-partition problem. The idea of the protocol is to assign states $0, 1, \ldots, n-1$ to n agents one by one and then regard an agent with state s as a member of the $(s \mod k)$-th group. One may

Algorithm 1 Asymmetric uniform k-partition protocol.

A variable at BS
  M: The state that the BS assigns next, initialized to 0
A variable at a mobile agent a:
  $S_a \in \{0, 1, 2, \ldots, P-1\}$: The agent state, initialized arbitrarily. Agent a belongs to the $(S_a \mod k)$-th group.

```
while a mobile agent a interacts with BS do
    if M ≤ S_a then
        S_a = M
        M = M + 1
    end if
end while
while two mobile agent a and b interact do
    if S_a = S_b and S_a < P − 1 then
        S_a = S_a + 1
    end if
end while
```

think that, to implement this idea, we can directly use a naming protocol [16], where the naming protocol assigns different states to agents by using P states if $n \leq P$ holds. Actually, if $n = P$ holds, the naming protocol assigns states 0, 1, …, $P-1$ to P agents one by one and hence it achieves uniform k-partition. However, if $n < P$ holds, the naming protocol does not always achieve uniform k-partition. For example, in the case of $(n-1)k < P$, the naming protocol may assign states 0, k, $2k$, …, $(n-1)k$ to n agents one by one, which implies that all agents are in the 0-th group.

Algorithm 1 shows a P-state asymmetric protocol for the uniform k-partition problem. In the protocol, the BS assigns states 0, 1, …, $n-1$ to n agents one by one. To do this, the BS maintains variable M, which represents the state the BS will assign next. The BS sets $M = 0$ initially, and increments M whenever it assigns M to an agent. Consider an interaction between the BS and an agent with state x. If x is smaller than M, the BS judges that it has already assigned a state to the agent, and hence it does not update the state. If x is M or larger, the BS assigns state M to the agent and increments M. When the BS assigns state x to an agent, there may exist another agent with state x because of arbitrary initial states. To treat this case, when two agents with the same state x interact, one transits to state $x+1$ and the other keeps its state x. By repeating such interactions, eventually exactly one agent has state x. By this behavior, the BS eventually assigns states 0, 1, …, $n-1$ to n agents one by one, and hence the algorithm achieves uniform k-partition.

As a result, we obtain the following theorem.

▶ **Theorem 18.** *Algorithm 1 solves the uniform k-partition problem. This means that, in the model with an initialized BS, there exists an asymmetric protocol with P states and arbitrary initial states that solves the uniform k-partition problem under weak fairness, where P is the known upper bound of the number of agents.*

▶ Remark 19. Interestingly, when P is odd, Algorithm 1 solves the uniform bipartition even if the number of agent states is $P-1$. Concretely, let $S_a \in \{1, 2, 3, \ldots, P-1\}$ be a set of agent states, and initialize variable M to 1. Then, Algorithm 1 converges to a configuration such that there exist two agents with state $P-1$ (and other states are held by exactly one agent). This is because, in the algorithm, the BS assigns $P-1$ agents to $P-1$ states one by

one, and, since the algorithm works under weak fairness, the remaining one agent shifts its state until state $P-1$. In the configuration, the difference in the numbers of *red* and *blue* agents is one. Moreover, every agent does not change its own state after the configuration. Hence, the uniform bipartition is solved. ◀

4.2 A Symmetric Protocol

In this subsection, we propose a $(P+1)$-state symmetric protocol for the uniform k-partition problem. We can easily obtain the protocol by a scheme proposed in [12]. In [12], a P-state symmetric protocol for the counting problem is proposed. The counting protocol assigns different states in $\{1,\ldots,n\}$ to n agents and keeps the configuration if $n < P$ holds. Hence, by regarding $P+1$ as the upper bound of the number of agents and allowing $P+1$ states, the protocol assigns different states in $\{1,...,n\}$ to n agents for any $n \leq P$. This implies that, as in the previous subsection, the protocol can achieve the uniform k-partition by regarding an agent with x as a member of the $(x \mod k)$-th group.

▶ **Theorem 20.** *In the model with an initialized BS, there exists a symmetric protocol with $P+1$ states and arbitrary initial states that solves the uniform k-partition problem under weak fairness, where P is the known upper bound of the number of agents.*

5 Results for Non-initialized BS

In this section, we show the impossibility with non-initialized BS. In the proof, we use ideas of the impossibility proof for the uniform bipartition protocol [32]. This work shows that, in the model with no BS, there is no protocol for uniform bipartition problem with arbitrary initial states under global fairness.

▶ **Theorem 21.** *In the model with non-initialized BS, no protocol with arbitrary initial states solves the uniform bipartition problem under global fairness.*

Proof. For contradiction, we assume such a protocol Alg exists. Moreover, we assume n is even and at least 4. We consider the following two cases.

First, consider population $A = \{a_0,\ldots,a_n\}$ of n agents and a non-initialized BS, where a_0 is the BS. For A, consider an execution $E = C_0, C_1, \cdots$ of Alg. From the definition of Alg, there exists a stable configuration C_t. Hence, both the number of *red* agents and the number of *blue* agents are $n/2$ at C_t. By the definition of a stale configuration, the color of agent a_i (i.e.,$f(s(a_i))$) never changes for any a_i $(1 \leq i \leq n)$ after C_t even if agents interact in any order.

Next, consider population $A' = \{a'_0\} \cup \{a'_i | f(s(a_i, C_t)) = red\}$, where a'_0 is the BS. For A', consider an execution $E' = C'_0, C'_1, \cdots$ of Alg from the initial configuration C'_0 such that $s(a'_i, C'_0) = s(a_i, C_t)$ holds for any $a'_i \in A'$. Note that, since we assume a non-initialized BS, even the BS can have $s(a_0, C_t)$ as its initial state. Since all agents are *red* at C'_0, some agents must change their colors to reach a stable configuration. This implies that, after C_t in execution E, agents change their colors if they interact similarly to E'. This is a contradiction. ◀

6 Conclusion

In this paper, we clarify solvability of the uniform bipartition with arbitrary initial states under weak fairness in the model with an initialized BS. Concretely, for asymmetric protocols, we show that P states are necessary and sufficient to solve the uniform k-partition problem

under the assumption, where P is the known upper bound of the number of agents. For symmetric protocols, we show that $P + 1$ states are necessary and sufficient under the assumption. Moreover, these upper and lower bounds can be applied to the k-partition problem under the assumption. There are some open problems as follows:

- Are there some relations between the uniform k-partition problem and other problems such as counting, leader election, and majority?
- What is the time complexity of the uniform k-partition problem?

References

1. Dan Alistarh, James Aspnes, David Eisenstat, Rati Gelashvili, and Ronald L Rivest. Time-space trade-offs in population protocols. In *Proc. of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2560–2579, 2017.
2. Dan Alistarh, James Aspnes, and Rati Gelashvili. Space-optimal majority in population protocols. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2221–2239. SIAM, 2018.
3. Dan Alistarh and Rati Gelashvili. Polylogarithmic-Time Leader Election in Population Protocols. In *Proc. of the 42nd International Colloquium on Automata, Languages, and Programming*, pages 479–491, 2015.
4. Dan Alistarh, Rati Gelashvili, and Milan Vojnović. Fast and exact majority in population protocols. In *Proc. of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 47–56, 2015.
5. Dana Angluin, James Aspnes, Melody Chan, Michael J Fischer, Hong Jiang, and René Peralta. Stably computable properties of network graphs. In *Proc. of International Conference on Distributed Computing in Sensor Systems*, pages 63–74, 2005.
6. 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.
7. Dana Angluin, James Aspnes, and David Eisenstat. A simple population protocol for fast robust approximate majority. *Distributed Computing*, 21(2):87–102, 2008.
8. Dana Angluin, James Aspnes, David Eisenstat, and Eric Ruppert. The computational power of population protocols. *Distributed Computing*, 20(4):279–304, 2007.
9. Dana Angluin, James Aspnes, Michael J Fischer, and Hong Jiang. Self-stabilizing population protocols. In *International Conference On Principles Of Distributed Systems*, pages 103–117. Springer, 2005.
10. James Aspnes, Joffroy Beauquier, Janna Burman, and Devan Sohier. Time and Space Optimal Counting in Population Protocols. In *Proc. of International Conference on Principles of Distributed Systems*, pages 13:1–13:17, 2016.
11. James Aspnes and Eric Ruppert. An introduction to population protocols. In *Middleware for Network Eccentric and Mobile Applications*, pages 97–120, 2009.
12. Joffroy Beauquier, Janna Burman, Simon Claviere, and Devan Sohier. Space-optimal counting in population protocols. In *Proc. of International Symposium on Distributed Computing*, pages 631–646, 2015.
13. Joffroy Beauquier, Julien Clement, Stephane Messika, Laurent Rosaz, and Brigitte Rozoy. Self-stabilizing counting in mobile sensor networks with a base station. In *Proc. of International Symposium on Distributed Computing*, pages 63–76, 2007.
14. Petra Berenbrink, Dominik Kaaser, Peter Kling, and Lena Otterbach. Simple and efficient leader election. In *1st Symposium on Simplicity in Algorithms (SOSA 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.
15. Andreas Bilke, Colin Cooper, Robert Elsaesser, and Tomasz Radzik. Population protocols for leader election and exact majority with O (logˆ 2 n) states and O (logˆ 2 n) convergence time. *arXiv preprint arXiv:1705.01146*, 2017.

16 Janna Burman, Joffroy Beauquier, and Devan Sohier. Brief Announcement: Space-Optimal Naming in Population Protocols. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, pages 479–481. ACM, 2018.

17 Shukai Cai, Taisuke Izumi, and Koichi Wada. How to prove impossibility under global fairness: On space complexity of self-stabilizing leader election on a population protocol model. *Theory of Computing Systems*, 50(3):433–445, 2012.

18 Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, and Eric Ruppert. When birds die: Making population protocols fault-tolerant. *Distributed Computing in Sensor Systems*, pages 51–66, 2006.

19 David Doty and David Soloveichik. Stable leader election in population protocols requires linear time. *Distributed Computing*, 31(4):257–271, 2018.

20 Leszek Gasieniec, David Hamilton, Russell Martin, Paul G Spirakis, and Grzegorz Stachowiak. Deterministic Population Protocols for Exact Majority and Plurality. In *Proc. of International Conference on Principles of Distributed Systems*, pages 14:1–14:14, 2016.

21 Leszek Gasieniec, Grzegorz Stachowiak, and Przemyslaw Uznanski. Almost logarithmic-time space optimal leader election in population protocols. *arXiv preprint arXiv:1802.06867*, 2018.

22 Leszek Gasieniec 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.

23 Taisuke Izumi. On space and time complexity of loosely-stabilizing leader election. In *Proc. of International Colloquium on Structural Information and Communication Complexity*, pages 299–312, 2015.

24 Tomoko Izumi, Keigo Kinpara, Taisuke Izumi, and Koichi Wada. Space-efficient self-stabilizing counting population protocols on mobile sensor networks. *Theoretical Computer Science*, 552:99–108, 2014.

25 Anissa Lamani and Masafumi Yamashita. Realization of Periodic Functions by Self-stabilizing Population Protocols with Synchronous Handshakes. In *Proc. of International Conference on Theory and Practice of Natural Computing*, pages 21–33, 2016.

26 Satoshi Murata, Akihiko Konagaya, Satoshi Kobayashi, Hirohide Saito, and Masami Hagiya. Molecular robotics: A new paradigm for artifacts. *New Generation Computing*, 31(1):27–45, 2013.

27 Yuichi Sudo, Fukuhito Ooshita, Taisuke Izumi, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. Logarithmic Expected-Time Leader Election in Population Protocol Model. *arXiv preprint arXiv:1812.11309*, 2018.

28 Yuichi Sudo, Fukuhito Ooshita, Hirotsugu Kakugawa, Toshimitsu Masuzawa, Ajoy K Datta, and Lawrence L Larmore. Loosely-Stabilizing Leader Election with Polylogarithmic Convergence Time. In *22nd International Conference on Principles of Distributed Systems (OPODIS 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

29 Tomoki Umino, Naoki Kitamura, and Taisuke Izumi. Differentiation in Population Protocols. *6th workshop on biological distributed algorithms(BDA)*, 2018.

30 Hiroto Yasumi, Naoki Kitamura, Fukuhito Ooshita, Taisuke Izumi, and Michiko Inoue. A Population Protocol for Uniform k-partition under Global Fairness. *International Journal of Networking and Computing*, 9(1):97–110, 2019.

31 Hiroto Yasumi, Fukuhito Ooshita, and Michiko Inoue. Uniform Partition in Population Protocol Model under Weak Fairness. *arXiv preprint arXiv:1911.04678*, 2019.

32 Hiroto Yasumi, Fukuhito Ooshita, Ken'ichi Yamaguchi, and Michiko Inoue. Constant-space population protocols for uniform bipartition. *the 21st International Conference on Principles of Distributed Systems*, 2017.

33 Hiroto Yasumi, Fukuhito Ooshita, Ken'ichi Yamaguchi, and Michiko Inoue. Space-optimal population protocols for uniform bipartition under global fairness. *IEICE TRANSACTIONS on Information and Systems*, 102(3):454–463, 2019.

Split and Migrate: Resource-Driven Placement and Discovery of Microservices at the Edge

Genc Tato
Univ Rennes, Inria, CNRS, IRISA, France

Marin Bertier
Univ Rennes, Inria, CNRS, IRISA, France
INSA Rennes, France

Etienne Rivière
UCLouvain, Belgium

Cédric Tedeschi
Univ Rennes, Inria, CNRS, IRISA, France

Abstract

Microservices architectures combine the use of fine-grained and independently-scalable services with lightweight communication protocols, such as REST calls over HTTP. Microservices bring flexibility to the development and deployment of application back-ends in the cloud.

Applications such as collaborative editing tools require frequent interactions between the front-end running on users' machines and a back-end formed of multiple microservices. User-perceived latencies depend on their connection to microservices, but also on the interaction patterns between these services and their databases. Placing services at the edge of the network, closer to the users, is necessary to reduce user-perceived latencies. It is however difficult to decide on the placement of *complete* stateful microservices at one specific core or edge location without trading between a latency reduction for some users and a latency increase for the others.

We present how to dynamically deploy microservices on a combination of core and edge resources to systematically reduce user-perceived latencies. Our approach enables the split of stateful microservices, and the placement of the resulting splits on appropriate core and edge sites. Koala, a decentralized and resource-driven service discovery middleware, enables REST calls to reach and use the appropriate split, with only minimal changes to a legacy microservices application. Locality awareness using network coordinates further enables to automatically migrate services split and follow the location of the users. We confirm the effectiveness of our approach with a full prototype and an application to ShareLatex, a microservices-based collaborative editing application.

2012 ACM Subject Classification Information systems → Distributed storage; Information systems → Service discovery and interfaces; Computer systems organization → Cloud computing

Keywords and phrases Distributed applications, Microservices, State management, Edge computing

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.9

Acknowledgements We thank the anonymous reviewers for their comments. This work was partially funded by the Belgian FNRS project DAPOCA (33694591) and partly supported by the Inria Project Lab program Discovery (`http://beyondtheclouds.github.io/`).

1 Introduction

Modern interactive applications combine a front-end running on client devices (e.g. in their web browser) with a back-end in the cloud. Collaborative *editing* applications, in which multiple users concurrently make changes to the same document, such as Google Docs, Microsoft Office 365, and ShareLatex, are good examples of such interactive applications. Quality of experience for users of such applications depends on low latencies between an action of one client and its visibility by other clients.

© Genc Tato, Marin Bertier, Etienne Rivière, and Cédric Tedeschi;
licensed under Creative Commons License CC-BY
23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller; Article No. 9; pp. 9:1–9:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

A solution to enable fast request-response latencies between the front-end and the back-end of a collaborative application is to deploy part of the back-end at the edge, i.e. on computing resources that are closer and accessible with low latencies from the front-end.

It is often challenging to adapt an application to make use of edge resources. Software monoliths typically require massive re-engineering to support a deployment on multiple sites, as they base the collaboration between their constituents on shared memory or common databases. Service-Oriented Architectures (SOAs) on the other hand present desirable features for this adaptation, by splitting the features of the application into independent services and decoupling service location and naming.

Microservices are a popular approach to SOAs [9, 35] adopted by many large-scale companies [15, 17]. Features of the back-end are handled by fine-grained services communicating through lightweight protocols, such as publish/subscribe or event stores [8]. The most common form of interaction between microservices is the use of point-to-point calls to Representational State Transfer (REST) APIs provided over HTTP.

We are interested in this work in the adaptation of microservices applications towards a joint deployment on core resources, e.g. in some cloud datacenter, and edge resources, e.g. at micro-clouds located in the same metropolitan-area network as the clients. Our objective is to reduce latencies between user actions and their visibility by other users.

We target collaborative editing applications based on microservices. We demonstrated in our previous work [25] that ShareLatex, an open source and microservices-based application for collaboratively editing LaTeX documents, could benefit from reduced user-perceived latencies thanks to a *static* core/edge deployment of its microservices. This previous work considers however the placement of *entire* services onto different sites, which may lead to trading latency reduction for some users for latency increases for the others. It also does not consider the adaptation of this placement based on the actual location of the application users.

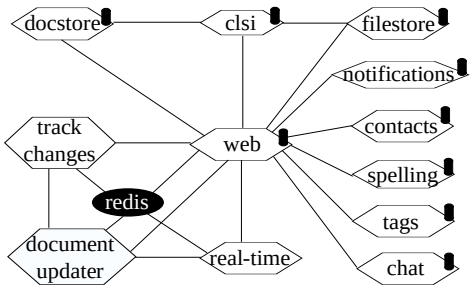
Contributions

We consider in this paper the *dynamic* placement and migration of microservices in core/edge deployments. We leverage the use in modern microservices applications of resource-centric REST APIs and NoSQL databases partitioned by a single primary key. This allows us to *split* microservices, and create independent instances responsible for a partition of the original service's data. These splits, deployed at different edge locations, can then handle requests for specific partitions of the service data, accessed by close-by users. We demonstrate our ideas with ShareLatex (**§2**).

Our first contribution is the support for splitting and multi-site placement of microservices. We detail how the state of a microservice can be partitioned, and how the resulting splits can be dynamically deployed on different core and edge sites (**§3**).

Our second contribution is the middleware support for the decentralized and dynamic *discovery* of microservice splits. We build on Koala [26], a lightweight Distributed Hash Table (DHT) for decentralized cloud infrastructures. We enable the transparent redirection of calls based on resource identifiers present in HTTP Uniform Resource Identifiers (URIs), also supporting the *relocation* of microservices splits. This allows adapting compatible legacy microservices applications for hybrid core/edge deployments with minimal effort (**§4**).

Our third contribution is a locality-driven policy conducting the *creation* and *migration* of microservices splits between the core and the edge, and between edge sites themselves, allowing to seamlessly adapt to the location of the users. This policy estimates latencies using network coordinates [13], enabling the automatic selection of the most appropriate site



Service	Description	Service	Description
1. docstore	CRUD ops on tex files	8. track-changes	History of changes
2. filestore	CRUD ops on binary files	9. real-time	Websocket server
3. clsi	Compile project	10. notifications	Notifications between users
4. contacts	Manage contacts	11. document-updater	Maintain consistent document state
5. spelling	Spell checking	12. web	User interface and service hub
6. chat	Chat service	Redis (db)	DB (Pub/Sub) for dynamic data
7. tags	Folders, tags	MongoDB (db)	DB for internal static data

Figure 1 ShareLatex architecture (left) and list of constituents (right).

for the services splits used by a group of collaborative users, with the goal of achieving better response times (**§5**).

We demonstrate our ideas on the ShareLatex application, using a representative core-edge network topology and measuring the impact of latencies at the level of the application front-end. Our results indicate that Koala and redirection layers induce only minimal overheads, while the dynamic placement of microservices splits enables users in different regions to access the same application with greatly reduced latencies (**§6**).

Finally, we present related work (**§7**) and conclude (**§8**).

2 ShareLatex and its core/edge deployment

ShareLatex is a collaborative application allowing users (e.g. students, researchers or writers of technical documentation) to concurrently edit a LaTeX project. It features a web-based editor with spellchecking and auto-completion, facilities for compiling and producing papers, and tools for the collaboration between writers, such as an address book and a chat service.

Responsiveness is a key element of the perceived quality of service in ShareLatex. For instance, a team of researchers could collaborate on the writing of algorithm pseudocode. Changes made by one researcher must be visible with no undue delay by the others, and changes must propagate as fast as possible to the reference document stored in the back-end to avoid concurrency hazards.

The ShareLatex back-end uses 12 microservices and a database, `Redis`, shared by four of them (Figure 1).[1] The `web` provides the front-end to the client browser and acts as an API gateway to other services. User actions (cursor changes, new text, etc.) are propagated by `web` to the `real-time` service using WebSockets. The `real-time` service then sends them to `document-updater` which is responsible for maintaining a consistent order of modifications. This dynamic state of the project is stored in `Redis`, and periodic snapshots are pushed to the `docstore` (text files) and `filestore` (binary files). Figure 1 details the ShareLatex architecture and its services.

Core servers are typically hosted in a centralized data center, while edge servers are distributed and closer to the users. In our previous work [25], we demonstrated that ShareLatex can benefit from a *static* placement of its services on a combination of core and edge

[1] Note that using a shared database does not fully comply with the microservices architectural pattern, where all state should be encapsulated in services. Yet, such compromises with the norm are found in many microservices-based applications. We prefer taking them into account rather than heavily modifying the legacy application code.

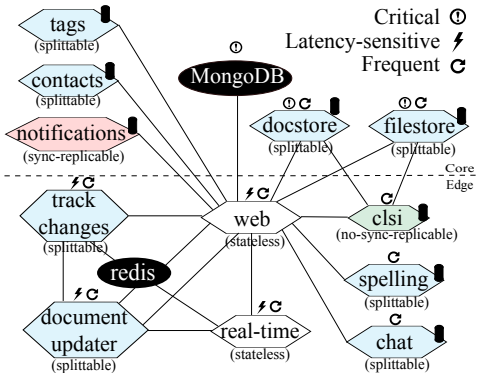


Figure 2 Static ShareLatex deployment as suggested in [25].

servers, closer to clients collaborating on a document. We build on our previous contribution, which requires only minimalistic modifications to the configuration and deployment scripts of ShareLatex, and no changes to the application code. The most significant modification performed in our previous work is the disassembly of the `web` service implementation from its database. This was necessary as `web` acts as an API gateway and *must* be deployed at the edge, but it also features a global database of information about users, which is queried infrequently. These queries can be done remotely to a database in the core, with minimal performance penalty.

The static core and edge placement of services of Figure 2 follows the recommendations argued in our previous work [25]: `web`, `real-time`, `document-updater` and `Redis` should be deployed on an edge site. Due to the coupling of `track-changes` with `Redis`, this service must be deployed alongside to avoid remote calls, even if it does not influence perceived latencies as much. The `clsi`, `spelling` and `chat` services can also be deployed at the edge, with a moderate but positive impact on perceived latencies. This placement resulted in lower latencies for operations impacting the most the user experience, at the cost of increasing latencies for operations that require interactions between services at the edge and services remaining in the core.

3 Splitting microservices

While some microservices may be *stateless*, most of them need to internally store and query data. A stateful microservice is typically implemented as a business-logic tier combined with a database. The choice of the appropriate database is specific to each microservice, leading to what is sometimes called a *polyglot* architecture. Figure 1 represents the presence of a database inside each service using a small black database symbol. In the unmodified ShareLatex, only `real-time` is a stateless service. All other services are stateful, including `document-updater` and `track-changes` which use the common `Redis` database. With the decoupling of `web` from its database (as depicted in Figure 2), this service is also stateless and uses remote calls to a MongoDB service.

A key property of SOA and therefore of microservices is the ability to independently *scale in* and *out* the business-logic tier and the database [17]. For the former, new *instances* may be created and deleted *on the fly*, e.g. using deployment middleware such as Kubernetes [7] and a scaling policy [28]. Elastic scaling is difficult to realize with relational databases, and microservices state may grow to large sizes requiring the ability to scale out storage to a large number of servers. NoSQL options with such horizontal scaling abilities are therefore a favored choice in many microservices applications.

NoSQL databases such as key/value stores or document stores, partition the data using a unique primary key. We observe that very often, accesses to the database by the business-logic tier for a query only read and write a *limited* and *identifiable* subset of keys. The identification of this subset typically depends on the characteristics of the query, and in particular on its *object*. It results that the state of the service, i.e. the content of the database, may be *partitioned* in such a way that keys that are accessed together for any future service requests belong to the same *partition*. This enables in turn the possibility to create multiple instances of the service, each equipped with one of the partitions. We call these services hosting independent partitions of the database *service splits*. A service that supports splitting is a *splittable* service.

Not all services are splittable. Some may require operations (e.g., Map/Reduce queries, scans, etc.) that operate on the entire content of the database. In some cases, it is not possible to identify a mapping between requests characteristics and partitions, e.g. when calls may use object keys generated at runtime or read from the database itself. These services are therefore only *replicable*: It is only possible to make *complete* copies of the service and its state. When these copies must be kept in sync for the well-functioning of the application, the service is *sync-replicable*. When operating on divergent copies does not impact, or impacts only marginally, the well-functioning of the application, provided that users systematically use the same copy, the service is *no-sync-replicable*.

The analysis of ShareLatex code results in the following categorization of services, also reflected in Figure 2.[2] The `notifications` service is *sync-replicable*, while `clsi`, handling the compilation, is *no-sync-replicable*: compilations across projects do not require consistent updates. The `web` service was initially *sync-replicable*, but the decoupling of its database makes it stateless. All other stateful services –a majority of them– are *splittable*. This means that their state (content of the services databases, but also the content of the shared `Redis` database) can be partitioned, and that partitions can be deterministically identified for any query. The object of the query, that allows identifying the partition of service state, and therefore the appropriate service split, is the specific writing *project* that the user is editing. In other words, the state of ShareLatex splittable services at the bottom of Figure 2 can be partitioned based on the project identifier, resulting in splits able to handle requests for a specific subset of projects. Such splits can then be deployed at the edge, and serve requests from close-by users accessing one of these projects.

The implementation of splitting requires support from the database embedded in splittable microservices, to be able to bulk load and store data partitions between an existing service and a newly created split. This support depends on the database API but does not pose implementation difficulties. For ShareLatex, we built minimalistic APIs enabling this for the `Redis` and `MongoDB` databases.

Our goal is to support the *dynamic* creation of service splits and their deployment over a combination of core and edge resources. This requires both appropriate *middleware support mechanisms* enabling the discovery and redirection of calls between microservices in a transparent manner, and appropriate *adaptation policies* to decide at runtime when and where to create splits, and when and where to migrate an existing split if its current location is not optimal. We cover these two aspects in the two following sections.

[2] This identification of services classes and partitions was performed manually, but did not represent a particularly difficult task in the case of ShareLatex. Automated or semi-automated service class identification and partitioning are beyond the scope of this paper, but we intend to explore these directions in our future work.

4 Discovering and redirecting to microservice splits

We now present the mechanisms that support the dynamic deployment of service splits on multiple sites. Our focus in this section is on the proper functioning of the system during and after service splitting and migration operations. We present the policies triggering these operations in the next section.

Our support middleware serves two purposes: Firstly, it enables the *discovery* of services and splits, and the live modification of their placement (§4.1). Secondly, it enables the *redirection* of point-to-point calls between source and destination services, ensuring that the core service or its appropriate split is reached (§4.2).

4.1 Discovery of microservice splits with Koala

Each service is initially associated with one instance in the core (the *core service*), responsible for its full state. Split and migrate operations dynamically update the list of splits for each service. Service discovery, therefore, requires the maintenance of an *index* of existing services, together with their current lists of splits. Every such split is associated with a list of object identifiers, for which this split is the only one able to process queries. This index must remain *strongly consistent*: At any point in time, there must be a single core service or split that can answer a query for a given object, and it must be impossible for two clients of the service under the same object to use different splits concurrently.

Service registries based on replicated databases updated using consensus (e.g., using etcd [11] or ZooKeeper [18]) are adapted for datacenter deployments with low network latencies. In our target context of distributed sites, centralizing the index would result in unacceptable overheads. We favor instead a decentralized design, supporting the caching and lazy revocation of split-to-site associations. This service is distributed, with an instance running at the core and at each of the edge sites.

Service discovery requests contain the name of the service, and for splittable services, the identifier of the query *object*. For ShareLatex splittable services, this object is the *project identifier*, that allows identifying the appropriate service state partition. Service discovery requests can be addressed to any of the sites.

The service index is implemented as a Distributed Hash Table (DHT), in which each node stores a subset of the index, partitioned using consistent hashing. Index elements are accessed using a primary key. Each node is responsible for a *range* of these keys. An overlay enables requests to deterministically reach the responsible node using greedy routing (each node in the path selects amongst the nodes it knows the closest to the destination). Typical DHT designs actively maintain all overlay links through the exchange of explicit overlay construction messages. In this work, we rely on Koala [27], a DHT that creates overlay links in a lazy manner, by piggybacking overlay construction messages over existing application traffic. This design choice enables to create more overlay links for routes in the overlay that are more frequently used for index reading requests, and minimize maintenance costs for seldom-used links. This is beneficial for workloads that are highly local, which is expected from service requests in one single application and to a relatively limited number of services (e.g. up to a few hundred).

Indexing

We keep two global indexes in Koala, an index of Objects, and an index of Splits. Figure 3 shows an example of the local subset of these indexes maintained by one Koala node. A Koala node is *responsible* for maintaining the authoritative and strongly consistent entry for a

Object ID	Location	Responsibility	Split group
Object 1	local	YES	[Service 1 - Split 1, Service 2 - Split 1]
Object 2	local	6-8	[Service 1 - Split 1]
Object 3	5-2	YES	-

Service name	Split ID	Location	Responsibility	IP	Port
Service 1	**Split 1**	local	YES	x.x.25.1	3001
Service 2	**Split 1**	local	6-8	x.x.25.2	3002
Service 3	**Split 1**	9-7	YES		
Service 3	**Split 2**	5-2	YES		

Figure 3 Indexes stored at some Koala DHT node: Objects table (left) and Splits table (right). Primary keys are in boldface.

number of index items, falling in its key responsibility range. It also maintains *local* resources, objects and splits, that are *hosted* on the corresponding edge site. A Koala node may have local resources for which it is not responsible or be responsible for resources that are not local. This design enables the creation of resources on a different node than the one that the DHT assigns for the corresponding entry index, while maintaining a single node in charge of this index entry and allowing atomic modifications. Lookups follow multiple hops in the overlay, until the *responsible* node is found, leading to one last hop to the node where the entry is *local* (if different). Nodes hosting locally a resource access it without involving the responsible node.

Discovery

A local split can only be reached by proxying through the local Koala instance.[3] The discovery of the appropriate site for an incoming service request proceeds in two phases. First, the Objects table in the DHT is queried to establish whether there exists a split of at least one service under that object. This information is stored in the *split group* for that object. If there is no entry for the object, or if there is no entry for the specific service in the split group, the request must go to the core. Second, the Koala node responsible for the split is located using the Splits table, using both the service name and split number as the key. This requires reaching the Koala node that is *responsible* for that key and then reaching the Koala node where that split is *local*.

For instance, on the node whose local subset of the index is represented by Figure 3, a request to Service 2 for Object 1 will be immediately sent to Service 2's Split 1 hosted locally. A request for Object 3 will be redirected in one hop to Koala node of identifier 5-2, to read its split group. A request for Object 4, not present in the local state, requires a multi-hop routing in the Koala overlay to request its service group.

Caching

Looking up service discovery entries in the DHT for *every* service call is too expensive. We implement *caching*: results of index lookup are kept at the local Koala node and reused. Stale cache entries are discarded in a lazy fashion. We leverage the fact that all requests must go through the *local* Koala node, e.g. on the edge site where the split actually runs. After the migration to a new site, queries based on stale cached information will arrive at the Koala node at the *previous* local location of the split. This node simply informs the origin, which invalidates related cache entries and falls back to a regular lookup.

[3] Allowing uncontrolled connections from outside of the edge site might not be possible due to network constraints, or not desirable for security reasons. The local Koala node acts, therefore, as an API gateway for all local service splits.

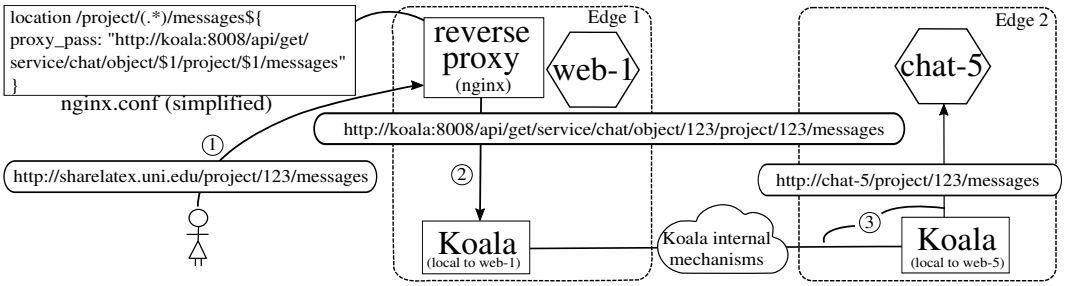


Figure 4 Example of REST call redirections in ShareLatex.

Migration

The *migration* of an existing split, or the creation of a new split, follows four phases. Firstly, an instance of the service is bootstrapped if none already exists at the destination edge site, or it is selected among existing instances, but it does not hold state or service requests. Secondly, a new entry in the Splits table is created to announce the existence of the new split. It does not contain a location yet. The split group for all corresponding objects is updated to indicate the temporary unavailability of the split. Service requests will block at the lookup request stage, and back off for a random time duration. Thirdly, the new instance receives the partition of the data from the source service or split. Finally, the Koala entry for the split is updated to reflect the location of the new *local* site for that split, and the split groups for all corresponding objects are updated. This allows request services to resume, using the new split location.

4.2 Transparent redirection of REST service calls

Modifying legacy microservices applications to directly make use of Koala APIs to discover and call services and splits would require an important effort. Instead, we leverage the fact that the objects of queries are accessible in the URIs of REST service calls. Indeed, REST being a resource-centric approach to designing interfaces, calls are made, typically over HTTP, to an explicit resource given in the request URI. We implement the transparent redirection of calls by extracting the object from this URI. Then, the local Koala node queries for the existence of a split for that object and the requested service. The request URI is transformed using rewriting rules to reach either the original core service, or the Koala node on the edge site where the split runs.

The implementation of the redirection is as follows. It is illustrated for a call in ShareLatex in Figure 4. We use the high-performance web server `nginx` as a reverse proxy for calls from, and to, local services. In ShareLatex, this includes the `web` service that serves as an API gateway for the user frontend. The reverse proxy translates the original request from the unmodified ShareLatex, to a request to the local Koala node. The discovery process detailed before establishes that there exists a split for that service that must serve the request. In the example of Figure 4, the `web` service on the Edge 1 site calls the `chat` service. The object "123", the project identifier, is extracted from the call URI. Koala then determines that the service split is on the Edge 2 site. The request is redirected to `chat` service in that site, where the call is handled by Koala.

5 Splits creation and migration policy

The creation of service splits and their migration between sites obey an adaptation *policy*. This policy must determine *what* service to split, *when* these split decisions are made and *where* to (re)deploy the splits. Its goal is to ensure that user-perceived latencies in the application are minimized.

What service to split?

The first aspect of the policy is application-dependent and results from the analysis of the interactions between its microservices. A set of *splittable* services, and not necessarily all of them, must be tagged for a preferential deployment at the edge. This aspect of the ShareLatex policy builds upon our previous results [25] (§2). Microservices that lie in the bottom part of Figure 2 are tagged for edge deployment. All other services always remain in the core.

When should splits happen?

There are two situations where a split may be formed: When a new object is created, and when latencies to the core are too high. The first option is sufficient for the ShareLatex policy: The creation of a new project leads to the immediate creation of all corresponding splits.

Where should splits go?

This aspect of the policy is twofold: Firstly, we must ensure that splits are created on a site (core or edge) close to the first user of the corresponding object. Secondly, we must adapt this placement when the chosen site is no longer the most adequate for the *current set* of users of that object.

This requires the ability to evaluate network latencies. Active probing of latencies (e.g. using ICMP packets) is impractical and unscalable. We combine two mechanisms to enable probe-less estimations. Firstly, we enforce that users always connect to the *geographically closest* site.[4] The location of a client is that of its connection site. Secondly, latencies between sites are estimated using Network Coordinates (NCs). We use Vivaldi [13] to compute NCs. Each site is represented by a d-dimensional point. These points positions evolve following a process similar to a spring-mass relaxation, based on observed latencies for *actual* message exchanges, and Euclidean distances eventually approximate latencies.

The ShareLatex policy enforces that the initial version of an object, and the corresponding splits, be hosted by the connection site of the first user. Each site collects for its local splits, a history of the NCs of the sites forwarding client calls. Periodically (every 5 minutes, or 100 requests, whichever comes first, in our implementation), the policy determines whether migration of the splits for each hosted object is necessary. Several users access a project, from different sites and with different frequencies. The ideal location of the splits for that project can be represented as a point in the NCs space. We define this point as the Center of Mass (CoM) for that object. It is the geometric average of the connection sites' NCs, weighted by the number of accesses from their clients. If there exists a site whose NC is closer to the CoM, the policy triggers a migration of all splits for that object to this new site.

[4] The list of core and edges sites IP is publicly known. Clients use an IP-to-location service (e.g. `www.iplocation.net`) and choose the geographically closest site.

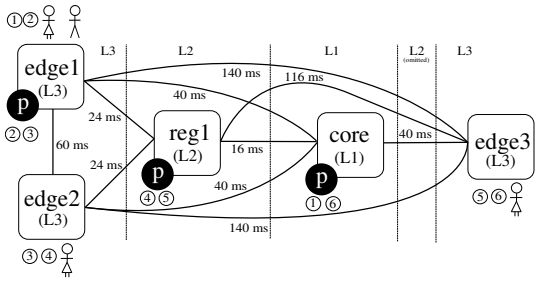


Figure 5 Topology and first experiment setup.

6 Evaluation

We evaluate the split and migrate principles with a full prototype, combining Koala, `nginx` reverse proxies, Docker CE for bootstrapping containers on the core and edge sites, and ShareLatex as the application.

Our evaluation aims at answering the following research questions: (i) Is the approach able to reduce perceived latencies for users of the application? (ii) Can the policy successfully migrate splits between edge sites when users' locations change? (iii) Is the overhead of using Koala and proxying acceptable?

We consider the three-layer (L1-L3) hierarchical topology shown in Figure 5. Its characteristics are derived from information obtained from an Internet Service Provider in the EU [23]. Layer L1 consists of the *core* site, L2 of regional sites (*reg1*) and L3 of edge sites (*edge1*, *edge2* and *edge3*). We deploy each site on a node of the Grid'5000 [5] testbed. Each node features 2 Intel Xeon E5-2630 v3 CPUs and 128GB of RAM. We emulate latencies between sites using the `tc` (traffic control) tool. Note that *reg1* is treated as an edge site, and that we ignore latencies between users and sites, and model their mobility by enforcing that they connect to a specific (closest) site. We use Network Coordinates (NCs) in $d = 2$ dimensions for ease of presentation, although a higher dimensionality (e.g. $d = 5$) would yield better estimations. Latencies are measured at the level of the instrumented ShareLatex frontend. We emulate the activity of users using the `Locust` [1] load testing tool, which allows describing programmatically the behavior of users as a list of actions and their respective occurrence frequencies.

6.1 Adaptation and split migrations for moving users

Our first experiment evaluates the ability of our approach to adapt the location of the splits for single a ShareLatex project, and the impact this has on latencies. We consider a project p shared by two equally active users, one stationary and one who changes her location continuously. Each user performs one operation every second, adding a new character to the text. The user-perceived latency is measured from the moment the text is updated by one user to the moment the update appears in the screen of the other user.

Figure 5 presents the experiment setup. Figure 6 presents the evolution of the average perceived latency for the two users, and Figure 7 presents the evolution of the CoM of the project. Circled numbers in all figures show the sequence of operations.

We follow three phases. In each phase, users are assigned to connection sites, and we observe the triggering and impact of the adaptation and resulting split migration decisions. Initially, both users are closer to *edge1* and therefore connect to that site. The latency for updating the text (50 ms) is roughly the RTT between *edge1* and *core*, plus the processing

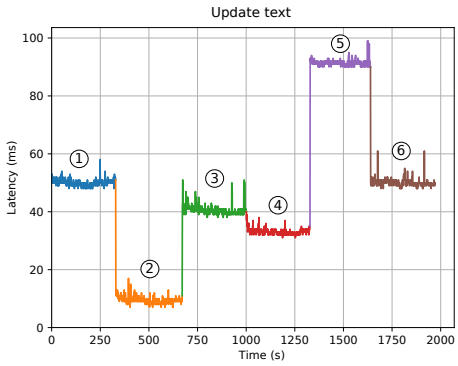


Figure 6 Evolution of *text update latencies* when migrating splits to follow a project CoM.

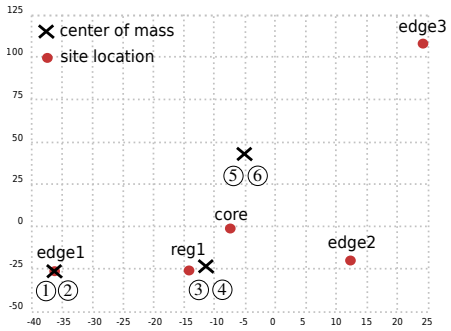


Figure 7 Evolution of Network Coordinates and CoMs when migrating splits.

Table 1 Distribution of projects, users and ideal site placements.

Project	p1	p2	p3	p4	p5	p6	p7	p8	p9	p10
Users	u1, u5, u6	u2	u1, u3	u1, u4	u5, u6	u6	u4, u7	u5, u8, u9	u5, u8, u9	u8, u10
User locations	e1, e2, e2	e1	e1, e1	e1, e1	e2, e2	e2	e1, e2	e2, e3, e3	e2, e3, e3	e3, e3
Ideal site(s)	e2, **r1**, e1	e1	e1	e1	e2	e2	r1, e1, **e2**	**e3**, core, e2	e3, **core**, e2	e3

time, of 40 ms and 10 ms respectively (① in Figure 6). Given that all requests for project p originate from the Koala instance on *edge1*, that location is also the CoM (① in Figure 7), and therefore the policy decides to split and migrate all tagged services to this site (②). The latency drops to slightly over the processing time. In a second phase, we move one of the users to *edge2* while the service splits for the project are still in *edge1* (③). This results in an increase in latencies. When it next triggers, the adaptation policy decides to migrate the service splits to *reg1* which is closer to the new CoM for the project (④). In the third phase, we move the user of *edge2* and connect it to *edge3* (⑤). The service splits are still in *reg1*, which results in high latencies. Again, the adaptation policy triggers and orders the migration of splits to the closest site to the CoM (⑥). The *core* happens to be the best compromise to serve the two users connected to *edge1* and *edge3*. This experiment shows that the policy is effective in splitting and migrating a single project according to its user locations, for a positive impact on perceived latencies.

6.2 Evolution of splits distributions

This second experiment shows how the split and migrate principles allow shifting the load from the core servers to edge servers while following the location of the most active users in a *collection* of ShareLatex projects. All services are initially only in *core*. We consider 10 users and 10 projects. Each project is edited by 1, 2 or 3 users. The two first lines of Table 1 show the mapping between users and projects. The third line indicates the (static) user locations for each project.

We model the activity of users to represent work sessions. During one hour and a half, every user randomly picks one of their assigned projects and edits if for a random duration of 2 to 10 minutes. The project CoM evolves to follow the location(s) of the currently active user(s). The fourth line of Table 1 indicates the possible ideal location(s) for the project splits, calculated offline.

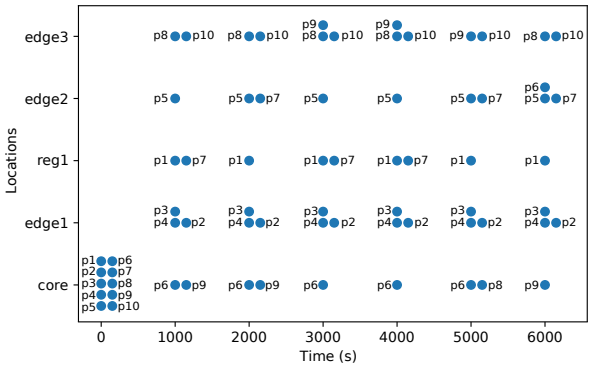


Figure 8 Evolution of splits placements.

We monitor the location of the service splits for the different projects, taking snapshots every 1,000 seconds. We run this experiment until the projects with a single ideal site placement reach this destination. Figure 8 presents these snapshots and the location of the service slices for the 10 projects. Projects whose ideal site is unique, such as *p2*-*p5* and *p10*, have the corresponding service slices migrated to these sites correctly and immediately. Projects with multiple ideal sites see their slices periodically migrate between these sites, following the currently active user(s). For instance, splits for *p7* move between *reg1* and *edge2*, while splits for *p8* and *p9* move between *edge3* and *core*. The final site is highlighted in boldface in Table 1. This experiment shows that the split and migrate mechanisms and the adaptation policy for ShareLatex allow dynamically moving microservices close to the users, based on the used resources (*projects* in ShareLatex).

6.3 Overheads of Koala and redirections

In this final experiment we evaluate the costs and overheads of the mechanisms enabling transparent call redirections. To isolate the overhead we compare a centralized setting where everything is deployed in the *core*, corresponding to the original ShareLatex model, with a one-edge-site setting where requests are redirected from this edge site to the core by Koala. Figure 9 presents this setup. We use a 50 ms latency between edge and core sites.

In both settings, the service split that responds to the user request is in *core*. In the centralized setting the request is first sent to the `web` core service and then forwarded to the right service directly, while in the second setting the request goes first through the local `web` split. This proxies the request to the Koala instance on *edge1*, which in turn forwards it to the Koala instance in *core* who then calls the service.

We distinguish three kinds of requests, two HTTP REST calls and one WebSocket request. For the REST calls, we consider a call to `tags`, for which splitting is disallowed (①), and a call to `chat`, which is splittable using the project identifier as the object (②). The WebSocket request updates the text (writing) ③. It is also a project-specific request and must reach the corresponding split of the `document-updater` service.

We expect a slightly higher overhead for redirections to split services compared to non-split ones. For non-split services, a single interaction with Koala is required (follow ①). For split services, two interactions are necessary: one to locate the object and one to redirect to the correct split (follow ② and ③).

The operation latencies times of the three requests with and without the redirection are shown in Figure 10. We consider two cases for the redirection: without and with caching.

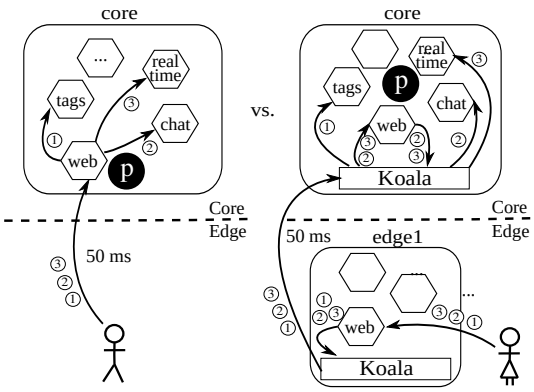


Figure 9 Setup for the experiment evaluating the overheads of Koala and redirections.

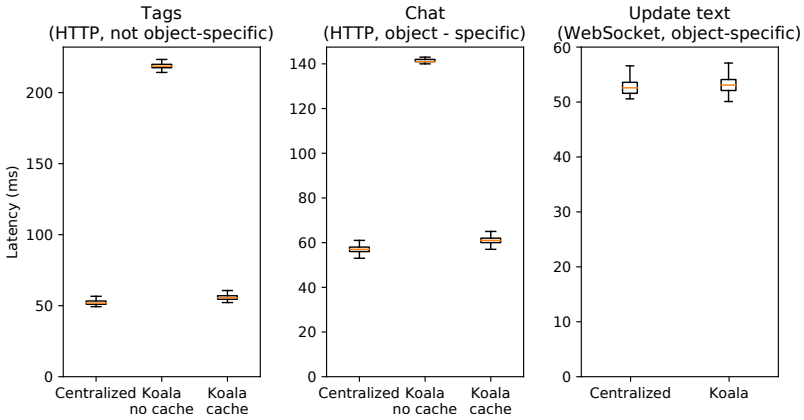


Figure 10 Evaluation of the overheads of Koala and redirections.

When the cache is disabled, lookups on the Koala DHT can require multiple hops between sites and incur a significant and unpredictable penalty. With caching, this penalty is only paid for the first access or after a migration invalidates the cached information. WebSocket requests occur on an established connection, therefore caching does not apply.

Figure 10 presents the distribution of latencies for the three operations and for 500 requests each. We observe a similar performance between the centralized setting and the setup using caching. The median overhead of proxying through the local edge site is ≈ 3 ms for the non-split service and ≈ 4 ms for the split one. For WebSockets operations this difference is smaller, ≈ 1 ms, which can be explained by the fact that this protocol is more lightweight than HTTP. Disabling caching leads to significant overheads as every operation leads to lookups in the DHT, bouncing between the core and edge Koala instances. This experiment shows that the latency impact of proxying through the edge is likely to be negligible compared to the gain of using locally-deployed services splits.

7 Related work

Previous research advocates to revisit the SOA paradigm for supporting service-based applications deployed in edge cloud platforms [19]: In light of the increase of the number of services at the edge able to answer a specific query, service registration must take into account spatial coverage, and service discovery must take locality into account. Our contributions are a step in that direction.

The placement of applications on fog platforms has been an active research topic in the recent years. One target domain is IoT applications where data collected from connected objects must be processed on nearby resources [24, 34]. Stream processing is another application that benefits from deployments on a combination of core and edge resources. It explicits its communication patterns (i.e., the directed acyclic graph linking stream processing operators), which can be leveraged for optimal placement on edge resources [12]. The Balanced RePartitioning (BRP) [4] algorithm targets generic distributed cloud applications and devises online algorithms which find a good trade-off between communication and migration costs.

Our work is linked with the concept of *mobile edge clouds*, where users move and connect to nearby resources dynamically [30]. When the mobility of users is modeled using Markov stochastic decision processes, analytical frameworks allow devising close-to-optimal algorithms for automating service placement [31]. Other approaches advocate the use of genetic algorithms to gradually refine an allocation of services to the edge [33].

We note that all of the aforementioned work considers the placement (and in some cases the migration) of *full* instances of services. We are not aware of solutions proposing to split stateful microservices and support resource-based discovery. State splitting is used, in a different context, for the elastic scaling of publish/subscribe middleware [6].

Research on collaborative edition has focused on enabling correctness and performance, including in the presence of network issues. The Jupiter protocol [21, 32] and the RGA protocol [22] implement a *replicated list object* abstraction and define how to propagate updates to achieve convergence [3]. Our work is complementary: The responsiveness of replicated list object algorithms (i.e. the time between an update and its visibility at the other clients) is sensitive to the latency between client nodes and a coordination server.

Service discovery middleware solutions for data centers typically rely on strongly consistent, fully replicated stores maintaining the complete index of services instances and of their locations. SmartStack [2], used for example by the Synapse [29] microservices platform, is based on Apache ZooKeeper [18]. Similarly to Koala, Synapse instances provide local proxies to services, but each maintains a full copy of the index while Koala relies on a DHT and caching for scalability. Kubernetes [10] leverages etcd [11] for service discovery. Recent work [14] suggests to add support for network coordinates [13] to route requests based on network locality. Yet, service selection decision remains a centralized process unlike with Koala where it can happen at the edge. Eureka [20] is also centralized but introduces the notion of *read clusters* that can serve requests closer to the clients. Unlike lazy cache management in Koala, read clusters must be explicitly synchronized when the service index changes. *Write clusters* can also be replicated, but are only eventually consistent, which makes them ill-suited for implementing consistent service migration. Finally, Consul [16] supports deployment to multiple data centers, and use network coordinates for location-aware selection. Consul only uses consensus-based synchronization within each individual data center. Updates propagate lazily between data centers using gossip, preventing consistent service relocation *across* data centers.

8 Conclusion

We presented how microservices could be dynamically deployed on a combination of core and edge resources. Our approach leverages the possibility to *split* microservices for which partitions of the data can be used to answer subsets of service requests independently. The Koala middleware enables to transparently redirect requests to the appropriate split based on object information available in REST calls URIs. Migration policies enable a dynamic placement of microservices splits on edge sites, and as our evaluation with the ShareLatex application shows, allow following the users and reduce perceived latencies.

This work opens interesting perspectives that we intend to consider in our future work. First, we wish to explore the automation of the identification of splittable microservices, and the use of static and dynamic analysis techniques to infer the relation between objects and state partitions. Second, we intend to extend support middleware to support redirections with other forms of communication, such as publish/subscribe or event sourcing [8]. Finally, we would like to build tools to automatize the identification of placement policies based on dynamic observations of communications between microservices.

References

1. Locust: An open source load testing tool. `https://www.locust.io`.
2. Airbnb. SmartStack Service Discovery in the Cloud. `https://bit.ly/2SAvRHn`.
3. Hagit Attiya, Sebastian Burckhardt, Alexey Gotsman, Adam Morrison, Hongseok Yang, and Marek Zawirski. Specification and complexity of collaborative text editing. In *ACM Symposium on Principles of Distributed Computing*, PODC. ACM, 2016.
4. Chen Avin, Andreas Loukas, Maciej Pacut, and Stefan Schmid. Online balanced repartitioning. In *International Symposium on Distributed Computing*, DISC. Springer, 2016.
5. Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Pérez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec. Adding Virtualization Capabilities to the Grid'5000 Testbed. In *Cloud Computing and Services Science*, volume 367 of *Communications in Computer and Information Science*. Springer, 2013.
6. Raphaël Barazzutti, Thomas Heinze, André Martin, Emanuel Onica, Pascal Felber, Christof Fetzer, Zbigniew Jerzak, Marcelo Pasin, and Etienne Rivière. Elastic scaling of a high-throughput content-based publish/subscribe engine. In *34th International Conference on Distributed Computing Systems*, ICDCS. IEEE, 2014.
7. David Bernstein. Containers and cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.
8. Dominic Betts, Julian Dominguez, Grigori Melnik, Fernando Simonazzi, and Mani Subramanian. *Exploring CQRS and Event Sourcing: A journey into high scalability, availability, and maintainability with Windows Azure*. Microsoft patterns & practices, 2013.
9. Fabienne Boyer, Xavier Etchevers, Noël De Palma, and Xinxiu Tao. Architecture-Based Automated Updates of Distributed Microservices. In *International Conference on Service-Oriented Computing*, ICSOC. Springer, 2018.
10. Cloud Native Computing Foundation. Kubernetes. `https://kubernetes.io/`.
11. CoreOS. Etcd reliable key-value store. `https://coreos.com/etcd/`.
12. Alexandre da Silva Veith, Marcos Dias de Assuncao, and Laurent Lefevre. Latency-Aware Placement of Data Stream Analytics on Edge Computing. In *International Conference on Service-Oriented Computing*, ICSOC. Springer, 2018.
13. Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: A decentralized network coordinate system. In *ACM SIGCOMM Computer Communication Review*, volume 34, 2004.
14. Ali Fahs and Guillaume Pierre. Proximity-Aware Traffic Routing in Distributed Fog Computing Platforms. In *IEEE/ACM International Symposium in Cluster, Cloud, and Grid Computing*, CCGrid, 2019.
15. Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *24th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS. ACM, 2019.
16. HashiCorp. Consul. `https://www.consul.io/`.

17 Wilhelm Hasselbring and Guido Steinacker. Microservice architectures for scalability, agility and reliability in e-commerce. In *Workshops of the Intl. Conf. on Software Architecture*, ICSA Workshops. IEEE, 2017.

18 Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX Annual Technical Conference*, ATC, 2010.

19 Valérie Issarny, Georgios Bouloukakis, Nikolaos Georgantas, and Benjamin Billet. Revisiting service-oriented architecture for the IoT: a middleware perspective. In *International Conference on Service-Oriented Computing*, ICSOC. Springer, 2016.

20 Netflix. Eureka 2.0. `https://bit.ly/2Mcexda`.

21 David A Nichols, Pavel Curtis, Michael Dixon, John Lamping, et al. High-latency, low-bandwidth windowing in the Jupiter collaboration system. In *ACM Symposium on User Interface Software and Technology*, 1995.

22 Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. Replicated abstract data types: Building blocks for collaborative applications. *Journal of Parallel and Distributed Computing*, 71(3):354–368, 2011.

23 Sanhaji A. (Orange Labs Networks, France). Private communication, 2019.

24 Olena Skarlat, Matteo Nardelli, Stefan Schulte, Michael Borkowski, and Philipp Leitner. Optimized IoT service placement in the fog. *Service Oriented Computing and Applications*, 11(4), 2017.

25 Genc Tato, Marin Bertier, Etienne Rivière, and Cédric Tedeschi. ShareLatex on the Edge: Evaluation of the Hybrid Core/Edge Deployment of a Microservices-based Application. In *3rd Workshop on Middleware for Edge Clouds & Cloudlets*, MECC. ACM, 2018.

26 Genc Tato, Marin Bertier, and Cédric Tedeschi. Designing Overlay Networks for Decentralized Clouds. In *Int. Conf. on Cloud Computing Technology and Science*, CloudCom. IEEE, 2017.

27 Genc Tato, Marin Bertier, and Cédric Tedeschi. Koala: Towards Lazy and Locality-Aware Overlays for Decentralized Clouds. In *2nd IEEE International Conference on Fog and Edge Computing*, ICFEC, 2018.

28 Giovanni Toffetti, Sandro Brunner, Martin Blöchlinger, Florian Dudouet, and Andrew Edmonds. An architecture for self-managing microservices. In *AIMC Workshop*. ACM, 2015.

29 Nicolas Viennot, Mathias Lécuyer, Jonathan Bell, Roxana Geambasu, and Jason Nieh. Synapse: A Microservices Architecture for Heterogeneous-database Web Applications. In *10th European Conference on Computer Systems*, ACM EuroSys, 2015.

30 Shiqiang Wang, Rahul Urgaonkar, Ting He, Kevin Chan, Murtaza Zafer, and Kin K Leung. Dynamic service placement for mobile micro-clouds with predicted future costs. *IEEE Transactions on Parallel and Distributed Systems*, 28(4), 2016.

31 Shiqiang Wang, Rahul Urgaonkar, Murtaza Zafer, Ting He, Kevin Chan, and Kin K Leung. Dynamic service migration in mobile edge-clouds. In *IFIP Networking Conference*, 2015.

32 Hengfeng Wei, Yu Huang, and Jian Lu. Specification and Implementation of Replicated List: The Jupiter Protocol Revisited. In *22nd International Conference on Principles of Distributed Systems, OPODIS*, Leibniz International Proceedings in Informatics (LIPIcs), 2018.

33 Hongyue Wu, Shuiguang Deng, Wei Li, Min Fu, Jianwei Yin, and Albert Y Zomaya. Service selection for composition in mobile edge computing systems. In *International Conference on Web Services*, ICWS. IEEE, 2018.

34 Ye Xia, Xavier Etchevers, Loic Letondeur, Adrien Lebre, Thierry Coupaye, and Frédéric Desprez. Combining heuristics to optimize and scale the placement of iot applications in the fog. In *IEEE/ACM 11th Int. Conf. on Utility and Cloud Computing*, UCC, 2018.

35 Uwe Zdun, Elena Navarro, and Frank Leymann. Ensuring and assessing architecture conformance to microservice decomposition patterns. In *International Conference on Service-Oriented Computing*, ICSOC. Springer, 2017.

HaTS: Hardware-Assisted Transaction Scheduler

Zhanhao Chen
Palo Alto Networks Inc, Santa Clara, CA, USA
zhachen@paloaltonetworks.com

Ahmed Hassan
Lehigh University, Bethlehem, PA, USA
ahmed.hassan@lehigh.edu

Masoomeh Javidi Kishi
Lehigh University, Bethlehem, PA, USA
maj717@lehigh.edu

Jacob Nelson
Lehigh University, Bethlehem, PA, USA
jjn217@lehigh.edu

Roberto Palmieri
Lehigh University, Bethlehem, PA, USA
palmieri@lehigh.edu

Abstract

In this paper we present HaTS, a Hardware-assisted Transaction Scheduler. HaTS improves performance of concurrent applications by classifying the executions of their atomic blocks (or in-memory transactions) into scheduling queues, according to their so called conflict indicators. The goal is to group those transactions that are conflicting while letting non-conflicting transactions proceed in parallel. Two core innovations characterize HaTS. First, HaTS does not assume the availability of precise information associated with incoming transactions in order to proceed with the classification. It relaxes this assumption by exploiting the inherent conflict resolution provided by Hardware Transactional Memory (HTM). Second, HaTS dynamically adjusts the number of the scheduling queues in order to capture the actual application contention level. Performance results using the STAMP benchmark suite show up to 2x improvement over state-of-the-art HTM-based scheduling techniques.

2012 ACM Subject Classification Software and its engineering → Scheduling

Keywords and phrases Transactions, Scheduling, Hardware Transactional Memory

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.10

Funding This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-17-1-0367 and by the National Science Foundation under Grant No. CNS-1814974.

1 Introduction

Without reservation, in-memory transactions have experienced a significant growth in adoption during the last decade. Specifically, the advent of Hardware Transactional Memory (HTM) support in commodity processors [27, 7, 16, 30] has changed the way concurrent programs' execution is handled, especially in terms of performance advantages. Whether a multi-threaded application implements atomic blocks using locks or transactions, HTM can be exploited in both cases (e.g., using Hardware Lock Elision [26] in the former case or Restricted Transactional Memory [27] in the latter case) to accelerate its performance.

© Zhanhao Chen, Ahmed Hassan, Masoomeh Javidi Kishi, Jacob Nelson, and Roberto Palmieri;
licensed under Creative Commons License CC-BY
23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller; Article No. 10; pp. 10:1–10:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Hardware transactions are significantly faster than their software counterpart because they rely on the hardware cache-coherence protocol to detect conflicts, while Software Transactional Memory (STM) [18] adds a significant overhead of instrumenting shared memory operations to accomplish the same goal [9]. Relying on the cache-coherence protocol also makes HTM appealing for mainstream adoption since it requires minimal changes in hardware. However, this inherent characteristic of HTM represents an obstacle towards defining contention management and scheduling policies for concurrent transactions, which are crucial for both progress and fairness of HTM execution in the presence of conflicting workloads. In fact, most TM implementations achieve high concurrency when the actual contention level is low (i.e., few transactions conflict with each other). At higher contention levels, without efficient scheduling, transactions abort each other more frequently, possibly with a domino effect that can easily lead to performance similar to, if not worse than, sequential execution [22, 21, 8].

A *Contention Manager* (CM) is the traditional, often encounter-time technique that helps in managing concurrency. When a transaction conflicts with another one, the CM is consulted to decide which of the two transactions can proceed. A CM collects statistics about each transaction (e.g., start time, read/write-sets, number of retries, user-defined parameters) and decides priorities among conflicting transactions according to the implemented policy. Schedulers are similar to CMs except that they may proactively defer the execution of some transactions to prevent conflicts rather than react to them. In both cases, performance is improved by decreasing abort rate and fairness is achieved by selecting the proper transaction to abort/defer [20, 31, 30].

The conflict resolution strategy of current off-the-shelf HTM implementations is provided entirely in hardware, and can be roughly summarized as follows:

- the L1 cache of each CPU-core is used as a buffer for transactional read and write operations[1];
- the granularity of conflict detection is the cache line; and
- if a cache line is evicted or invalidated, the transaction is aborted (reproducing the idea of read-set and write-set invalidation of STM [11]).

The above strategy thus implies a requester-wins contention management policy [6], which informally means that a transaction T_1 aborts another transaction T_2 if T_2 performed an operation on a memory location that is physically stored in the same cache line currently requested by T_1, excluding the case of two read operations, which never abort each other. Due to this simple policy, classical CM policies cannot be trivially ported for scheduling HTM transaction mainly because of two reasons. First, transactions are immediately aborted when one of the cache lines in their footprint is invalidated, which makes it too late for CM to avoid conflicts or manage them differently (e.g., by deciding which transaction is more convenient to abort). Second, it is hard to embed additional metadata to monitor transactions behavior, since all reads and writes executed within the boundaries of transactions are considered transactional, even if the accessed locations store metadata rather than actual data.

In this paper we introduce HaTS (*Hardware-assisted Transaction Scheduler*), a transaction scheduler that leverages the unique characteristics of HTM to accelerate scheduling in-memory transactions. To overcome the aforementioned limitations of HTM, HaTS neither aims at altering HTM's conflict resolution policy nor adds metadata instrumentation inside hardware transactions, but instead relies on it to relax the need for the scheduler to define a non-

[1] In some HTM implementation [27], reads can be logged on the L2 cache to increase capacity.

conflicting schedule. HaTS effectively arranges incoming transactions according to a set of metadata collected either at compilation time (leveraging developer's annotations) or at run time (after transactions commit/abort). HTM is then used to execute transactions concurrently while maintaining atomicity, isolation, and performance.

In a nutshell, HaTS works as follows. It uses a software classifier to queue incoming transactions with the goal of allowing only those transactions that do not conflict with each other to execute concurrently. The fundamental innovation of HaTS, which makes it practical, is that it works with incomplete or even erroneous information associated with incoming transactions. This is because even if the classifier erroneously allows conflicting transactions to run concurrently, HTM will inherently prevent them from committing (at least one of the conflicting transactions will abort). Therefore, misclassifications cannot impact the correctness of the transactional execution.

More in detail, HaTS offers a set of *scheduling queues* to group conflicting transactions. Membership of a queue is determined based on a single metadata object associated with each transaction, called *conflict indicator*. A conflict indicator might be provided by the programmer (e.g., the address of a contended memory location accessed transactionally) or computed by the system (e.g., transaction abort rate).

A queued transaction waits until the scheduler signals it when it becomes top-standing in its respective queue. When the transaction actually executes, the built-in HTM takes care of possible conflicts with transactions dispatched from other queues due to misclassification, which also includes the case where no conflict indicator is provided.

Another key feature of HaTS is that it adapts the number of scheduling queues based on a set of global statistics, such as the overall number of commits and aborts. This adaptation significantly improves performance in two common, and apparently dual cases. On the one hand, since conflict indicators are mainly *best effort* indicators, a single per transactions conflict indicator will not suffice when the application workload is highly conflicting. For this reason, HaTS reduces the number of queues when the overall conflict level increases, *enforcing transactions with different conflict indicators to be executed sequentially*. On the other hand, if the overall conflict level significantly decreases, HaTS increases the number of queues in order to (re-)allow transactions with different conflict indicators to execute concurrently. Additionally, when conflict level remains low, it enables dispatching multiple transactions from the same queue simultaneously, *allowing transactions with the same conflict indicator to execute in parallel*. Our framework aims at adaptively converging on an effective configuration of scheduling queues for the actual application workload. By leveraging HTM and its built-in atomicity guarantees, transitions between different configurations do not entail stalling transaction executions.

We implemented HaTS in C++ and integrated it into the software framework of SEER [13]. We contrasted HaTS performance against Hardware Lock Elision (HLE) [26], Restricted Transactional Memory (RTM) [27], Software-assisted Conflict Management (SCM) [1], and SEER itself. We used the STAMP suite as our benchmark and we used a testbed of four-socket Intel platform with HTM implemented through TSX-NI. Results, including speedup over the sequential non-instrumented code and two abort rate metrics, show that HaTS outperforms competitors in both high contention and low contention scenarios. This stems from both leveraging conflict indicators and performing dynamic adjustment of scheduling queues, which leads to a notable performance improvement (e.g., 2x speedup in execution time for `Kmeans` and a 50% improvement for the `Vacation` benchmarks).

The rest of this paper is structured as follows. We review previous work in Section 2 and the limitations of the current HTM implementations in Section 3. The design and implementation details of HaTS are presented in Section 4. We compare the performance of HaTS against state-of-art competitors in Section 5, and we conclude our paper in Section 6.

2 Related Work

TM schedulers can be classified based on whether they target STM systems [28, 15, 14, 5] or HTM systems [13, 25, 1, 4, 29, 31]. Although we deploy and test HaTS in an HTM-based environment, due to its performance advantages, HaTS can be deployed in STM systems as well.

Among the HTM-based schedulers, the closest one to HaTS is SEER [13]. SEER's main idea is to infer the probability that two atomic blocks conflict, basing its observation on the commit/abort pattern witnessed in the past while the two were scheduled concurrently. Thus, HaTS and SEER are similar in their best-effort nature: both of them do not require precise information on the pair of conflicting transactions nor on the memory location(s) where they conflict, which is the key to coping with the limitations of current HTM implementations.

HaTS differs from SEER in two core points. First, SEER focuses only on managing HTM limitations, and thus it schedules transactions based on their commit/abort patterns. On the other hand, HaTS is a generic scheduler that uses HTM to preserve atomicity and consistency, and thus it defines a generic *conflict indicator* object that can embed both online and offline metadata. Second, SEER adopts a fine-grained (pairwise) locking approach to prevent transactions that are more likely to conflict from running concurrently. HaTS replaces this fine-grained locking scheme with a lightweight queueing scheme that controls the level of conflict by increasing/decreasing the number of scheduling queues. Our experimental results in Section 5 show that this lightweight scheme results in a lower overhead in different scenarios, as the case of Vacation where the percentage of committed transactions in HTM is comparable with SEER's but overall application execution time is about 50% faster.

The work in [1] proposes a Software-assisted Conflict Management (SCM) extension to HTM-based lock elision (HLE) [26], where aborted transactions are serialized (using an auxiliary lock) and retried in HTM instead of falling back to a slow path with a single global lock. The main advantage of this approach is avoiding the lemming effect that causes new (non-conflicting) transactions to fall back to the slow path as well. As opposed to HaTS, SCM uses a conservative scheme where all aborted transactions are serialized without any further (even imprecise) conflict indicators, which limits concurrency. Moreover, SCM does not leverage the observed conflict pattern to proactively prevent conflicts in the future.

The idea of using scheduling queues to group conflicting transactions has been briefly discussed in [25], where authors introduced the concept of Octonauts. Octonauts uses statistics from transactions that committed in HTM to speculate over the characteristics of the associated transaction profile for future classification. HaTS is an evolution of Octonauts where a comprehensive software infrastructure, along with conflict indicators and a dynamic scheduling queuing techniques have been used to improve application performance.

A few other HTM-based schedulers were proposed prior to the release of Intel TSX extensions [27]. However, they either assume HTM implementations that are different from the currently deployed hardware [4, 29] or rely on a conservative single-lock-based serialization scheme similar to SCM [31].

STM-based schedulers rely on more precise information about transactions conflict. ProPS [28] uses a probabilistic approach similar to SEER but with a precise knowledge of the pair of conflicting transactions. Shrink [15] additionally uses the history of recently completed transactions' read/write-sets to predict conflicts. CAR-STM [14] and SOA [5] use per-core queues such that an aborted transaction is placed in the queue of the transaction that causes it to abort. The necessity of precise information represents an obstacle towards adopting such techniques in HTM-based systems.

3 Background: Scheduling Best-effort Hardware Transactions

HTM provides a convenient concurrent programming abstraction because it guarantees safe and efficient accesses to shared memory. HTM executes atomic blocks of code optimistically, and during the execution all read and write operations to shared memory are recorded in a per-transaction log, which is maintained in a thread-local cache. Any two operations generated by two concurrent transactions accessing memory mapped to the same cache line trigger the abort of one of the transactions. HTM is known to have limited progress guarantees [3, 8, 24]. To guarantee progress, all HTM transactions are guaranteed to commit after a number of retries as HTM transactions by exploiting the traditional software lock-based fallback path [27]. To implement that, hardware transactions check if the fallback lock is acquired at the beginning of their execution. If so, the transactional execution is retried; otherwise the execution proceeds in hardware and mutual exclusion with the fallback path is implemented by leveraging the strong atomicity property of HTM, which aborts any hardware execution if the fallback lock is acquired at any moment. To reduce the well-known lemming effect [10] in HaTS, a transaction is not retried in HTM until the global lock is released.

For simplicity, in the rest of the paper we refer to HTM execution as the above process, which encompasses hardware trials followed by the fallback path, if needed. It is important to note that, since HaTS does not assume a specific methodology to provide HTM with progress, more optimized alternative solutions [22, 8, 12] can be integrated into our HTM execution to improve performance even further.

The off-the-shelf HTM implementation only provides limited information about reasons behind aborted transactions, which makes it very hard for programmer to introduce modifications that would increase the likelihood for that transaction to commit. As a result, in the presence of applications with contention, HTM might waste many CPU cycles until a transaction can succeed by either retrying multiple times, or by falling back to a single global lock where the protected HTM execution can be relaxed in favor of the mutual exclusion implemented by the lock itself.

Contention management for practical transaction processing systems is often formulated as an online problem where metadata, in the form of statistics (e.g., actual access pattern), can be collected by aborted and committed transactions in order to fine-tune scheduling activities. However, this methodology cannot be directly ported to HTM-protected concurrent executions since HTM cannot distinguish between a cache line that stores actual application data, or scheduling metadata. Because of that, conflicting accesses to shared metadata executed by two concurrent hardware transactions may cause at least one of them to abort, even if at the semantic level no conflict occurred.

The above issues motivated us to design a transaction scheduler where HTM is exploited as-is, instead of providing software innovations or hardware extensions aimed at influencing the HTM conflict resolution mechanism [2], which likely lead to degradation of HTM effectiveness.

4 Hardware Transaction Scheduler

In this section we overview the two core components of HaTS, namely the transaction conflict indicator (Section 4.1) and the dynamic scheduling queues (Section 4.2), along with a description of the complete transaction execution flow (Section 4.3) and the details of how threads execution passes through the scheduling queues (Section 4.4).

Terminology. HaTS has a set of N concurrent queues, called *scheduling queues*. Each thread that is about to start a transaction (i.e., an atomic block) is mapped to one of those scheduling queues, and it starts executing only when HaTS dispatches it from that queue. Each scheduling queue has one (or more) *dispatcher thread*(s). As we detail later, the mapping between each transaction and its corresponding scheduling queue is based on the transaction's *conflict indicator* and the mapping is implemented using hashing. The overall transaction commit/abort statistics are collected by HaTS and recorded into a shared *knowledge base*. HaTS periodically consults the knowledge base to increase/decrease the number of scheduling queues or the number of dispatcher threads, dynamically.

4.1 Transaction Conflict Indicator

HaTS uses a so called *transaction conflict indicator*, provided as a parameter to `TM-BEGIN` in our experimental study, to represent in a compact way characteristics that affect the probability of aborting a hardware transaction due to conflict. Having this information is indeed powerful because it allows HaTS to group transactions that access the same system's hot spot in the same conflict queue, which saves aborts and increases throughput.

The transaction conflict indicator is an abstraction that can be deployed in many different ways. A simple and effective example of conflict indicator is the address of the memory location associated with the accessed system hot spot. As a concrete example in a real application, let us consider a monetary application where transactions work on given bank accounts. A transaction would use the address of the accessed bank account, which uniquely identifies that object (or memory location) in the system, as its conflict indicator. This way, although transactions might still (occasionally) conflict on other shared memory elements, HaTS will be able to prevent conflicts between accesses to the same bank account, which is one of the most contended set of objects in the system. Because of its effectiveness, in our evaluation study we focused on system hot spots as the transaction conflict indicator.

Other examples of transaction conflict indicators include:

- *Abstract data types of accessed objects:* transactions accessing (possibly different) objects of the same abstract data type will have the same conflict indicator. This represents a more conservative approach than our adopted (per-object) conflict indicator, and it can work better in workloads with higher contention levels.
- *HLE fallback lock(s):* if hardware transactions are used for lock elision (HLE) [26], the fallback paths of HTM transactions acquire the elided locks rather than a single global lock, as in RTM. Using this fallback lock as a conflict indicator, transactions that elide the same lock are grouped together.
- *Profile of aborted transactions:* HaTS's knowledge base can record the profile identification of aborted transactions within a window of execution, and group incoming invocations of those transactions using a single conflict indicator. This can significantly reduce abort rate because those transactions are more prone to conflict in the future as well. To avoid unnecessary serialization, the knowledge base can record only transactions aborted due to conflict, and exclude transactions aborted for any other reason (i.e., capacity and explicit aborts).
- *Core ID:* transactions running on the same physical core are assigned the same conflict indicator. This can be beneficial because multiple hardware threads running on the same physical core share the same L1 cache, which means that transactions concurrently invoked by those threads are more prone to exceed cache capacity and abort.

The last two points reflect similar ideas already used in literature in different ways [13, 1, 28, 15, 14, 5]. Although we refer to Section 2 for a detailed comparison of these approaches with HaTS, it is worth to mention here that HaTS's innovations relies on the fact that it deploys those ideas in an abstract way, using conflict indicators, which allows for a better scheduling management.

HaTS allows for the specification of a single conflict indicator per transaction. Although a single indicator might seem limited in terms of expressiveness, we adopt this approach because of the following reasons. First, there will always be a trade-off between the precision achieved by allowing multiple indicators and the additional cost needed to analyze them. Our decision is the consequence of an empirical study, that we excluded for space limitations, where we analyzed this trade-off. Second, the way HaTS adapts the number of scheduling queues (as detailed in the next section) is a dominant factor to manage contention that mitigates the effect of having imprecise, yet lightweight, conflict indicators. Finally, it is still possible to extend HaTS's infrastructure to support multiple indicators. For example, Bloom Filters can be used to compact multiple indicators and bit-wise operations (using either conjunction or disjunction operators) can be used to hash each bloom filter to the corresponding queue. As a future work, we plan to study the trade-off mentioned above; however, our current evaluation shows that even with a single indicator, HaTS outperforms existing approaches.

4.2 Dynamic Distribution of Scheduling Queues

Mapping transaction conflict indicators to scheduling queues is critical for achieving the goal of HaTS because it guarantees that transactions with the same conflict indicators are grouped in the same queue. However, using a static number of scheduling queues in such a mapping might lead to issues such as unbalancing, unfair scheduling, and poor adaptivity to some application workloads. For this reason, HaTS deploys a dynamic number of scheduling queues to cope with application workloads and effectively provide an elastic degree of parallelism. As we detailed in Section 4.3, this number is decided at run time according to the overall commit/abort statistics calculated in HaTS's knowledge base.

The following two examples clarify the need for having a dynamic set of scheduling queues. First, consider two incoming transactions with different conflict indicators. Since we have a finite number of scheduling queues, it is possible that those two transactions are mapped to the same queue. When the number N of queues is increased, the probability of mapping those transactions to the same queue decreases, and the level of parallelism increases. Second, consider a workload where transactions are highly conflicting so that the conflict indicator is not sufficient to capture all raised conflicts. In this case, decreasing the number of queues reduces parallelism and potentially reduces abort rate.

Adaptively changing the number of queues also covers more complex, yet not uncommon, cases. For example, it covers the case when transactions' data access pattern is hard to predict; therefore having a single conflict indicator per transaction may not be sufficient (e.g., when each transaction accesses multiple system hot spots). Also, it covers the cases when no effective conflict indicator can be defined but the workload experiences high abort rates due to other reasons (e.g., aborts due to false sharing of the same cache lines). Finally, it allows schedulers to temporarily disable the usage of conflicting indicator as a medium for grouping transactions, in favor of a random policy, without hindering performance.

As will be clear in Section 4.3, dynamically changing the number of scheduling queues neither introduces blocking phases nor trades off correctness, thanks to the underlying HTM.

4.2.1 Multiple Dispatchers

An interesting example that is not covered by the aforementioned policy is when transactions with the same conflict indicator (and hence grouped in the same queue) are actually able to execute concurrently. Although it may appear as an infrequent case, we recall that conflict indicators are best-effort indicators that can be imprecise. Also, since conflicts are raised at runtime according to the transaction execution pattern, it may happen that two conflicting concurrent transactions succeed to commit even if they run concurrently (e.g., one of them commits before the other one reaches the conflicting part).

HaTS addresses this case by allowing multiple dispatcher threads for a single scheduling queue. Similar to the way we increase/decrease the number queues, we use abort rate as an indicator to increase/decrease the number of dispatchers per queue. For additional fine-tuning, we allow programmers to statically configure the number of scheduling queues. In that sense, transactions with the same conflict indicator are executed in parallel only if the overall contention level is low.

4.3 Transaction Execution Flow

Transactional operations are executed directly by application threads, without relying on designated worker threads managed by HaTS. In fact, HaTS's role is to dispatch thread executions.

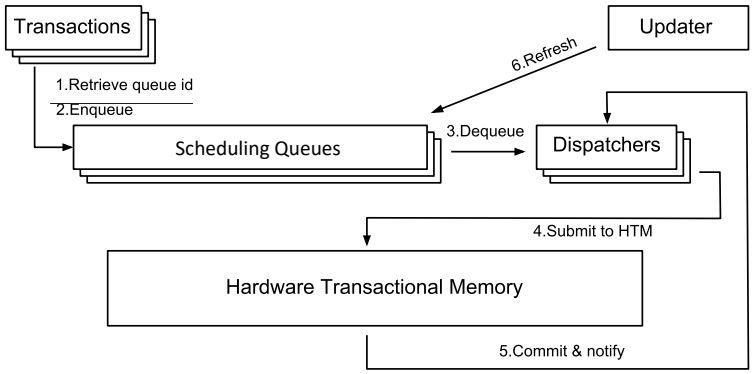


Figure 1 HaTS software architecture and high-level threads execution flow.

HaTS restricts transactions mapped to the same scheduling queue to run sequentially, while offloading the concurrency control handling to the underlying HTM engine. Figure 1 shows the execution flow of a transaction T executed by an application thread T_r. T_r first hashes the *conflict indicator* of T (using `module` N hashing, where N is the current number of scheduling queues) in order to find the matching scheduling queue Q (Step 1). After that, T_r effectively *suspends* its execution by enqueuing itself into Q (Step 2) and waiting until a dispatcher thread *resumes* its execution (Step 3).

HaTS provides one (or more) dispatcher thread T_Q per conflict queue Q. Each dispatcher thread resumes one waiting transaction execution at a time, in a closed-loop manner, meaning the next queued transaction execution is resumed only after the previous one is successfully committed. For the sake of fairness, each queue is implemented as a priority queue, where the priority of each transaction is proportional to the number of aborts its enclosing atomic block experienced in former executions (similar to the approach used in SEER [13] to infer the conflict pattern between atomic blocks.).

When a thread execution is resumed, the corresponding transaction starts to execute leveraging the HTM implementation (Step 4). During the hardware transaction execution, T_Q waits until T_r completes its transactional execution. After T_r's commit, T_Q takes control (Step 5) and performs two operations: it updates the *knowledge base* with the needed information (namely the number and types of aborts before committing); and it dispatches the next thread execution waiting in Q.

A special background thread, called *updater*, is used to dynamically change the number of scheduling queues depending upon the effectiveness of the parallelism achieved by the current scheduling queues configuration. To do so, the updater thread queries the knowledge base and decides, according to the transaction abort rate measured so far, whether the total number N of scheduling queues should be increased, decreased, or unchanged (Step 6). In our implementation, we adopt a simple hill-climbing approach similar to the one used in [17, 12]. Briefly, if the observed abort rate is greater (less) than the last observed rate, we decrease (increase) the number of queues by one. The maximum number of queues is set to the number of physical cores and the minimum is set to one. We also allow programmers to override this policy by setting a fixed number of scheduling queues in order to eliminate the overhead of this dynamic behavior, especially when the most effective configuration is known. Interestingly, as we show later in our experimental results, this simple approach pays off in most of the tested benchmarks. As a future work, we plan to investigate more complex approaches, such as collecting more detailed information (e.g., the types of aborts) an use reinforcement learning to reach better estimates.

Changing the scheduling queues configuration does not cause stalls of the transaction execution and does not affect execution correctness. This is a great advantage of leveraging HTM conflict detection. Let us consider two incoming transactions T_1 and T_2 that in the current scheduling queues configuration would map to the same scheduling queue Q_1. Let us assume that the scheduling queues configuration changes after T_1 is enqueued in Q_1 and before T_2 is enqueued. In this case, there is the possibility for T_2 to be mapped to another queue Q_2 in the new configuration, which ends up having T_1 and T_2 in two different queues (even though it might be the case they were both to be mapped to the same queue Q_2 in the new configurations). Although one can consider this scenario as an example of misclassification of incoming transactions due to ongoing change of N, safety cannot be affected because of leveraging the HTM execution. Even if those transactions are conflicting, they will be correctly serialized by (likely) enforcing one of them to abort and fallback to HTM's global locking phase.

4.4 Suspending/Resuming Executions with Scheduling Queues

As we mentioned in the previous section, a thread that wants to execute a transaction suspends and enqueues its execution until a dispatcher thread of the mapped queue resumes it. In order to synchronize this suspend/resume process, we use two synchronization flags[2], one handled by the application thread and the other handled by the dispatcher thread.

Figure 2 pictures the synchronization scheme between an application thread T_r performing a transaction T and a dispatcher thread T_Q responsible for handling the scheduling queue Q that matches T's conflict indicator. Numbers represent the sequence of operations completion.

When T_r wants to execute T, it creates a flag object L^{T_r} initialized to 0 and (atomically) enqueues it in Q, which effectively suspends its execution. After that, T_r spins until L^{T_r} is set to 1. When L^{T_r} becomes top standing in Q, T_Q dequeues it. Then, T_Q sets a flag

[2] Flags are implemented as `volatile` shared memory locations.

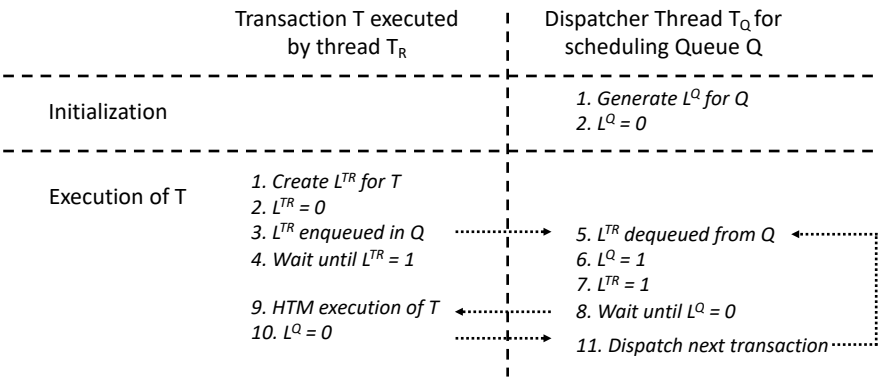


Figure 2 Synchronization between application threads and dispatcher threads. For simplicity, the example accounts for a single dispatcher thread per scheduling queue.

associated with Q, called L^Q, and also sets L^{T_r} to 1 (in that order). By setting L^{T_r}, T_r will be signaled to proceed with its HTM execution. By setting L^Q, T_Q is suspended until the completion of T by T_r. This suspension is implemented by spinning over the L^Q flag. When T is committed, T_r resets L^Q so that T_Q can dequeue the next thread execution waiting on Q. Note that T_Q is not notified if T is aborted and restarted for another HTM trial or if T's execution falls back to the software path. T_Q is resumed only after T's successful completion.

In our implementation we use simple flags to synchronize two threads (i.e., application thread and dispatcher thread) because we deploy one dispatcher thread for each scheduling queue. As introduced earlier, HaTS allows for multiple dispatcher threads per queue in order to cope with the case where the mapping between conflict indicators and scheduling queues is unnecessarily unbalanced, meaning many transactions, possibly with different conflict indicators, are grouped on the same scheduling queue. In the case where multiple dispatcher threads are deployed per conflict queue, the same synchronization scheme illustrated before applies, with the following differences. First, L^{T_r} flags should be atomically set (e.g., using a Compare-And-Swap operation) to synchronize between dispatcher threads. Also, multiple L^Q flags, one per dispatcher thread, are needed to signal each dispatcher thread that it may proceed to schedule the next transaction.

Scheduling queues are implemented in a lock-free manner [19] in order to speed up the thread execution's suspension step. Also, in the above description we simplified the presentation by saying that application threads and dispatcher threads spin over flags to suspend/resume their execution. In the actual implementation, threads yield their execution in order to let computing resources available so that the machine can be oversubscribed (if needed) to maximize CPU utilization.

5 Evaluation

HaTS is implemented in C++ and integrated into the software framework of SEER [13]. An advantage of using a unique software architecture for all competitors is that independent optimization of low-level components does not bias performance towards some implementation. In other words, the performance differences reported in our plots are due to the algorithmic differences between competitors.

Our goal is to assess the practicality of HaTS in standard benchmarks for in-memory transactional execution. Hence, we used STAMP [23], a benchmark suite of eight concurrent applications that span several application domains with different execution patterns. Due to

space limitations, we present in detail the results of two applications, `Kmeans` and `Vacation`, since they cover two different and important cases in which the characteristics of HaTS are highlighted. Then, we summarize the results with the other applications.

We compare HaTS against the scheduling techniques provided in the SEER framework, which are (in addition to SEER itself): Hardware Lock Elision (HLE) [26], Restricted Transactional Memory (RTM) [27], and Software-assisted Conflict Management (SCM) [1]. Shortly, SEER is the state-of-the-art probability-based scheduling mechanism that uses fine-grained (pairwise) locks to prevent transactions that are likely to conflict from executing concurrently. HLE transforms atomic blocks protected by locks into hardware transactions and its software fallback path is protected by the original lock itself. In STAMP, a single lock for each atomic block is deployed. RTM supports a configurable number of retries before falling back to a single global lock shared among all hardware transactions. SCM implements a scheduling technique that serializes the aborted transactions to decrease the chance of further aborts. In all implementations, except HLE, transactions try at most five times in hardware before migrating to the software fallback path.

Experiments were conducted using a multi-processor platform equipped with 4 Intel Xeon Platinum 8160 processors (2.1GHz, 24 cores per CPU). The machine provides 96 physical cores and a total of 768 GB of memory divided into 4 NUMA zones. In our experiments we ran up to 64 application threads to leave resources for dispatcher threads (one per queue) and the updater thread. The maximum number of scheduling queues is statically set to 30 prior execution, and we used the default operating system policy to map application threads to cores.

In Figure 3, we report for each application three performance metrics: (left column) speedup over sequential non-instrumented execution; (middle column) percentage of transactions committed through HTM; (right column) among those committed in HTM, percentage of transactions retried more than one time. Generally, the last two metrics are indicators of the scheduling effectiveness in reducing abort rate. The speedup metric is an indicator of whether such a reduction in abort rate is reflected in overall performance improvement or the scheduling overhead nullifies performance benefits. Performance at one application thread represent the slowdown of the sequential instrumented execution. All results are the average of 10 repeated tests.

The first application is `Kmeans`, a clustering algorithm that groups points into K clusters. Transactions are used by the application to synchronize the concurrent updates to the same cluster's center node. For this reason, we select the address of the center node updated by the transaction as its conflict indicator. We implemented that by passing the address of this center node as a parameter to STAMP's `TM-BEGIN` function. Our main observation is that identifying this conflict indicator allows HaTS to significantly reduce abort rate, reaching up to 1.5x reduction with respect to the closest competitor, and improve performance, reaching up to 2x better speedup over the closest competitor. SEER's probabilistic approach is the second best in terms of abort rate, which means that its approach is still able to capture conflicts, but not as effectively as using HaTS's conflict indicator. Moreover, SEER's speedup significantly decreases with higher number of threads, due to its locking overhead.

HLE does not perform well due to the lemming effect, which is visible as soon as few transactions fall back to locking. RTM is generally better than HLE due to its multiple retries in HTM before falling back to locking. SCM provides the worst performance. This is because the way SCM serializes conflicting transactions does not prevent new conflicts of incoming transactions, as opposed to the proactive scheduling approach, such as the one of HaTS and SEER. Also, the probability SCM executes non-conflicting transactions serially is higher than HaTS because it does not use any conflict indicators.

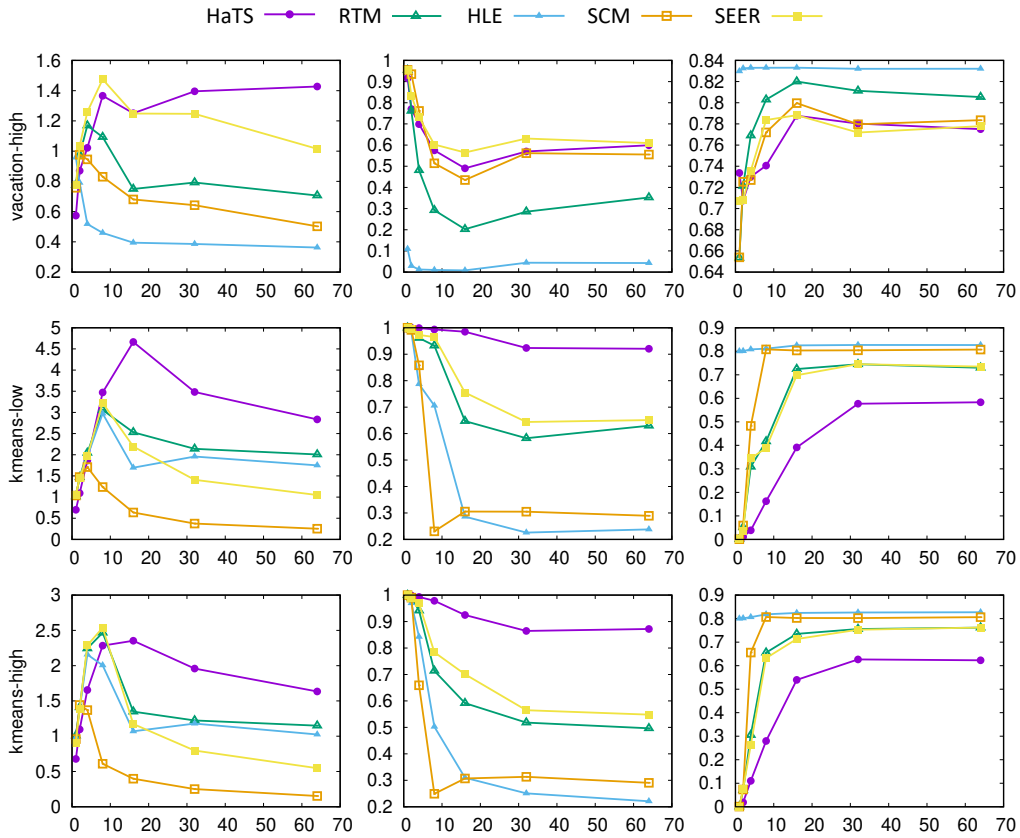


Figure 3 Results with applications of STAMP benchmark. For each row, the left plot shows speedup over sequential (non-instrumented) execution; center plot shows the ratio of committed transactions in HTM; the right plot shows the ratio of transactions retried more than one time. X-axis shows number of application threads.

The difference between the high and low configuration of Kmeans is mainly in the maximum achieved speedup. However, the patterns of the compared algorithms remain the same, which shows the capability of HaTS to gain performance even in relatively low-contention workloads.

The second application is `Vacation`, which simulates an online reservation system. Unlike `Kmeans`, most transactions in `Vacation` apply operations on a set of randomly selected objects, therefore with this pattern it is hard to identify a single conflict indicator per transaction. For that reason, we adopt an approach where each transaction uses a unique conflict indicator, with the exception of transactions that access a single customer object, where we use the customer ID as a conflict indicator. Our rationale behind this decision is that even if transactions are conflicting, HaTS's dynamic adjustment of scheduling queues reduces the level of parallelism and saves aborts. Indeed HaTS achieves an abort rate similar to SEER, and moreover it scales better than SEER.

Summarizing our results of the other STAMP benchmarks (Figure 4) `Intruder` and `Yada` give the same conclusions: the lightweight queuing approach in HaTS allows it to perform better than SEER, especially for high number of threads, due to the overhead of SEER's locking mechanism. `SSCA` and `Genome` are low-contention benchmarks, and their abort rates are very low even without scheduling or contention management. Hence, none of the compared

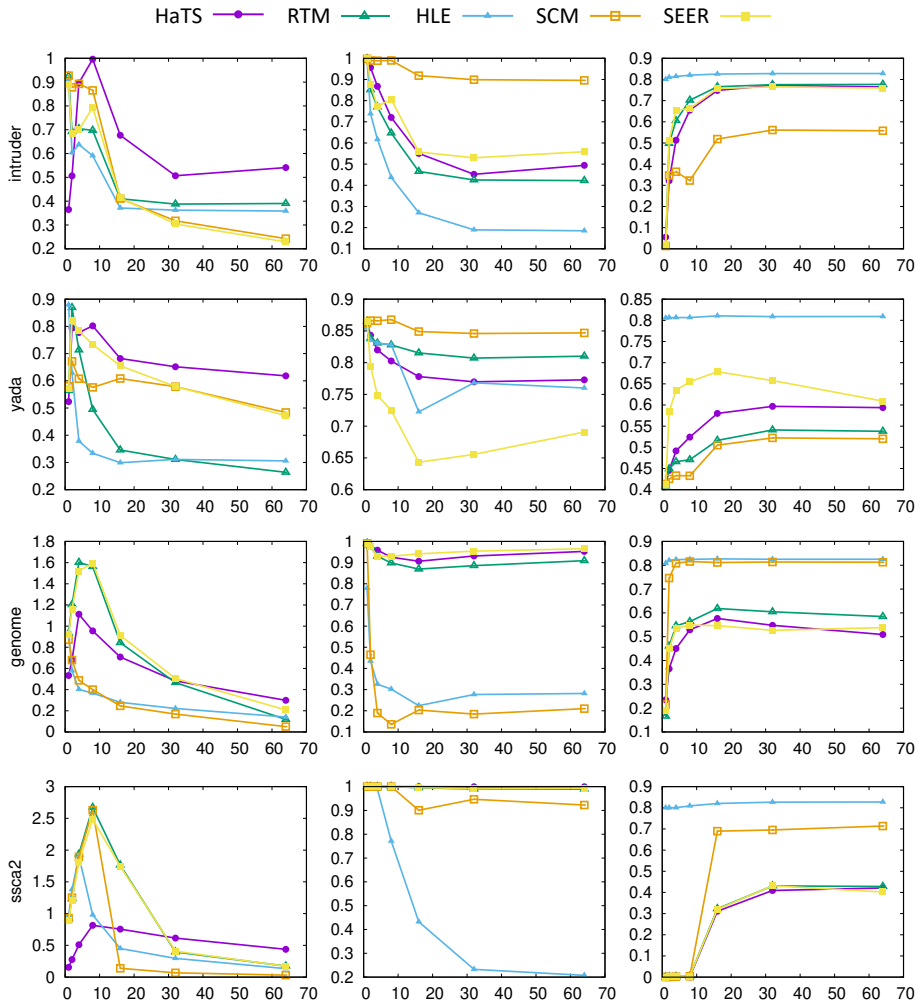


Figure 4 Performance results on the remaining applications of STAMP benchmark. X-axis shows number of application threads.

algorithms had a significant improvement over the others. However, HaTS maintains its performance better than others when the number of threads (and thus contention level) increases. We excluded `Bayes` and `Labyrinth` because it is known they provide unstable, and thus unreliable, results [13].

6 Conclusion

In this paper we presented HaTS, a Hardware-assisted Transaction Scheduler. HaTS groups incoming transactions into scheduling queues depending upon the specified conflict indicators. HaTS exploits the HTM conflict resolution to cope with the possibility of having erroneous conflict indicators or when conflict indicators are complex to identify. Results using the STAMP benchmark show that HaTS provides improvements in both high contention and low contention workloads.

References

1. Yehuda Afek, Amir Levy, and Adam Morrison. Software-improved Hardware Lock Elision. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 212–221, New York, NY, USA, 2014. ACM. doi:10.1145/2611462.2611482.
2. Dan Alistarh, Syed Kamran Haider, Raphael Kübler, and Giorgi Nadiradze. The Transactional Conflict Problem. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, pages 383–392, New York, NY, USA, 2018. ACM. doi:10.1145/3210377.3210406.
3. Dan Alistarh, Justin Kopinsky, Petr Kuznetsov, Srivatsan Ravi, and Nir Shavit. Inherent limitations of hybrid transactional memory. *Distributed Computing*, 31(3):167–185, June 2018. doi:10.1007/s00446-017-0305-3.
4. Mohammad Ansari, Behram Khan, Mikel Luján, Christos Kotselidis, Chris C. Kirkham, and Ian Watson. Improving Performance by Reducing Aborts in Hardware Transactional Memory. In *High Performance Embedded Architectures and Compilers, 5th International Conference, HiPEAC 2010, Pisa, Italy, January 25-27, 2010. Proceedings*, pages 35–49, 2010.
5. Mohammad Ansari, Mikel Luján, Christos Kotselidis, Kim Jarvis, Chris C. Kirkham, and Ian Watson. Steal-on-Abort: Improving Transactional Memory Performance through Dynamic Transaction Reordering. In *High Performance Embedded Architectures and Compilers, Fourth International Conference, HiPEAC 2009, Paphos, Cyprus, January 25-28, 2009. Proceedings*, pages 4–18, 2009.
6. Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware transactional memory. In *34th International Symposium on Computer Architecture (ISCA 2007), June 9-13, 2007, San Diego, California, USA*, pages 81–91, 2007.
7. Harold W. Cain, Maged M. Michael, Brad Frey, Cathy May, Derek Williams, and Hung Le. Robust Architectural Support for Transactional Memory in the Power Architecture. In *ISCA*, pages 225–236, 2013.
8. Irina Calciu, Justin Gottschlich, Tatiana Shpeisman, Gilles Pokam, and Maurice Herlihy. Invyswell: A Hybrid Transactional Memory for Haswell's Restricted Transactional Memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques*, PACT '14, 2014.
9. Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software Transactional Memory: Why Is It Only a Research Toy? *Queue*, 6(5):40:46–40:58, September 2008. doi:10.1145/1454456.1454466.
10. Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early Experience with a Commercial Hardware Transactional Memory Implementation. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 157–168, New York, NY, USA, 2009. ACM. doi:10.1145/1508244.1508263.
11. Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In Shlomi Dolev, editor, *Distributed Computing*, volume 4167 of *Lecture Notes in Computer Science*, pages 194–208. Springer Berlin Heidelberg, 2006. doi:10.1007/11864219_14.
12. Nuno Diegues and Paolo Romano. Self-Tuning Intel Transactional Synchronization Extensions. In *11th International Conference on Autonomic Computing*, ICAC '14. USENIX Association, 2014.
13. Nuno Diegues, Paolo Romano, and Stoyan Garbatov. Seer: Probabilistic Scheduling for Hardware Transactional Memory. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pages 224–233, New York, NY, USA, 2015. ACM. doi:10.1145/2755573.2755578.
14. Shlomi Dolev, Danny Hendler, and Adi Suissa. CAR-STM: scheduling-based collision avoidance and resolution for software transactional memory. In *Proceedings of the Twenty-Seventh Annual*

ACM Symposium on Principles of Distributed Computing, PODC 2008, Toronto, Canada, August 18-21, 2008, pages 125–134, 2008.

15 Aleksandar Dragojevic, Rachid Guerraoui, Anmol V. Singh, and Vasu Singh. Preventing versus curing: avoiding conflicts in transactional memories. In *Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing, PODC 2009, Calgary, Alberta, Canada, August 10-12, 2009*, pages 7–16, 2009.

16 Ricardo Filipe, Shady Issa, Paolo Romano, and João Barreto. Stretching the Capacity of Hardware Transactional Memory in IBM POWER Architectures. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pages 107–119, New York, NY, USA, 2019. ACM. doi:10.1145/3293883.3295714.

17 Ahmed Hassan, Roberto Palmieri, and Binoy Ravindran. Transactional Interference-Less Balanced Tree. In *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, pages 325–340, 2015.

18 Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM. doi:10.1145/165123.165164.

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

20 Junwhan Kim, Roberto Palmieri, and Binoy Ravindran. Enhancing Concurrency in Distributed Transactional Memory through Commutativity. In Felix Wolf, Bernd Mohr, and Dieter an Mey, editors, *Euro-Par 2013 Parallel Processing*, pages 150–161, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

21 Alexander Matveev and Nir Shavit. Reduced Hardware Transactions: A New Approach to Hybrid Transactional Memory. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, pages 11–22, New York, NY, USA, 2013. ACM. doi:10.1145/2486159.2486188.

22 Alexander Matveev and Nir Shavit. Reduced Hardware NOrec: A Safe and Scalable Hybrid Transactional Memory. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 59–71, New York, NY, USA, 2015. ACM. doi:10.1145/2694344.2694393.

23 Chi Cao Minh, Jaewoong Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 35–46, September 2008. doi:10.1109/IISWC.2008.4636089.

24 M. Mohamedin, R. Palmieri, A. Hassan, and B. Ravindran. Managing Resource Limitation of Best-Effort HTM. *IEEE Transactions on Parallel and Distributed Systems*, 28(8):2299–2313, August 2017. doi:10.1109/TPDS.2017.2668415.

25 Mohamed Mohamedin, Roberto Palmieri, and Binoy Ravindran. Brief Announcement: On Scheduling Best-Effort HTM Transactions. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pages 74–76, New York, NY, USA, 2015. ACM. doi:10.1145/2755573.2755612.

26 Ravi Rajwar and James R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual International Symposium on Microarchitecture, Austin, Texas, USA, December 1-5, 2001*, pages 294–305, 2001.

27 James Reinders. Transactional synchronization in haswell. *Intel Software Network. URL: http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/*, 2012.

28 Hugo Rito and João P. Cachopo. ProPS: A Progressively Pessimistic Scheduler for Software Transactional Memory. In *Euro-Par 2014 Parallel Processing - 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings*, pages 150–161, 2014.

29 Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Bhandari Aditya, and Emmett Witchel. TxLinux: using and managing hardware transactional memory in an operating system. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, pages 87–102, 2007.

30 Lingxiang Xiang and Michael L. Scott. Conflict Reduction in Hardware Transactions Using Advisory Locks. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pages 234–243, New York, NY, USA, 2015. ACM. doi:10.1145/2755573.2755577.

31 Richard M. Yoo and Hsien-Hsin S. Lee. Adaptive transaction scheduling for transactional memory systems. In *SPAA 2008: Proceedings of the 20th Annual ACM Symposium on Parallelism in Algorithms and Architectures, Munich, Germany, June 14-16, 2008*, pages 169–178, 2008.

Minha: Large-Scale Distributed Systems Testing Made Practical

Nuno Machado
Teradata, Madrid, Spain
INESC TEC, Porto, Portugal
nuno.machado@teradata.com

Francisco Maia
INESC TEC, Porto, Portugal
franciso.a.maia@inesctec.pt

Francisco Neves
INESC TEC, Porto, Portugal
U.Minho, Braga, Portugal
franciso.t.neves@inesctec.pt

Fábio Coelho
INESC TEC, Porto, Portugal
U.Minho, Braga, Portugal
fabio.a.coelho@inesctec.pt

José Pereira
INESC TEC, Porto, Portugal
U.Minho, Braga, Portugal
jop@di.uminho.pt

Abstract

Testing large-scale distributed system software is still far from practical as the sheer scale needed and the inherent non-determinism make it very expensive to deploy and use realistically large environments, even with cloud computing and state-of-the-art automation. Moreover, observing global states without disturbing the system under test is itself difficult. This is particularly troubling as the gap between distributed algorithms and their implementations can easily introduce subtle bugs that are disclosed only with suitably large scale tests.

We address this challenge with Minha, a framework that virtualizes multiple JVM instances in a single JVM, thus simulating a distributed environment where each host runs on a separate machine, accessing dedicated network and CPU resources. The key contributions are the ability to run off-the-shelf concurrent and distributed JVM bytecode programs while at the same time scaling up to thousands of virtual nodes; and enabling global observation within standard software testing frameworks. Our experiments with two distributed systems show the usefulness of Minha in disclosing errors, evaluating global properties, and in scaling tests orders of magnitude with the same hardware resources.

2012 ACM Subject Classification Computing methodologies → Distributed computing methodologies

Keywords and phrases Distributed software testing, Large scale distributed systems, Simulation

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.11

Funding This work is financed by National Funds through the Portuguese funding agency, FCT – Fundação para a Ciência e a Tecnologia within project: UID/EEA/50014/2019.

© Nuno Machado, Francisco Maia, Francisco Neves, Fábio Coelho, and José Pereira;
licensed under Creative Commons License CC-BY
23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller; Article No. 11; pp. 11:1–11:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Formal validation and verification tools are increasingly practical and of paramount importance in developing distribute algorithms. However, they work over models rather than actual runnable code [21, 4] and with assumptions that do not hold in practice [10]. In addition to outright bugs introduced by the translation from algorithm to runnable code, such as race conditions, a major problem lies in aspects that are abstracted by models and are revealed only in large scale tests [19]. A common problem is a situation in which the complexity of an operation (e.g., searching) is not apparent in testing as some data structure (e.g., a buffer) is mostly empty unless there is a large number of participants or with congestion, that never happens with small scale testing.

For example, the correctness proof of the accrual failure detector employed by Cassandra [15] assumed a negligible processing time of heartbeat messages. However, in practice, Cassandra uses variable length messages during membership changes in the cluster that may consume significant transmission and computation time. This mismatch between protocol design and implementation caused constant *flapping* problems in 500+ node deployments, preventing the cluster from stabilizing and scaling [6]. Flapping is a cluster instability problem where the status of the nodes is continuously switching between up and down.

Unfortunately, testing and debugging such systems is extremely challenging, mainly due to the following reasons:

Non-determinism and huge state space. Distributed executions often entail thousands of non-deterministic, concurrent events (e.g. message arrivals, node crashes, and timeouts) that, depending on the order in which they occur, cause the system to behave differently. Although most event sequences are correct, some incorrect timings of events result in severe damage, such as data corruption or system downtime [23]. In particular, previous work has shown that real-world distributed applications are especially prone to bugs stemming from *message races* [23, 26]. For instance, bug 2212 in ZooKeeper's issue repository reports a (rare) scenario in which a new node joining the cluster cannot become a leader due to a race condition between a message from the atomic broadcast protocol and one from the leader election protocol [7]. This bug causes a 3-node ZooKeeper cluster to stop working in the presence of a single failure, when in fact the existence of a majority of two nodes alive is sufficient for the system to operate.

Lack of resources for testing at scale. Distributed systems are typically developed to be deployed on a massive number of independent nodes (e.g. Cassandra [8], Hadoop [14], etc). Alas, as testing with close-to-production conditions can be prohibitively costly and time consuming, these applications are often debugged on small/medium-size deployments that prevent certain faulty behavior from manifesting [24]. Attempts to address scalability in testing environments include cloud computing and virtualization technologies [13, 38], but software validation is still the limiting factor in distributed software development [11].

Difficulty in checking global properties at runtime. In general, large-scale distributed protocols are designed in such a way that nodes make decisions based only on local information (or a partial view of the system). On the other hand, the correctness of these protocols often implies certain global properties or invariants to hold, which can be hard to verify without a globally-consistent snapshot of the system. For example, Pastry [33] is a distributed hash table whose routing algorithm requires each node to maintain a list with its physically closest

peers. Thus, to assess the correct execution of Pastry, one needs to first obtain the complete network overlay, then collect the neighbor lists from all nodes and, finally, compute the distances in both cases to check whether the references in the lists actually correspond to the closest peers.

Contribution. In this paper we propose Minha,[1] a framework that combines virtualization and simulation techniques for making large-scale distributed systems testing practical by providing a unique trade-off between scale and observation completeness. In particular, Minha addresses the problems introduced in the translation from algorithm to runnable code and message races, such as experienced in Cassandra [6] and Zookeeper [7], with two contributions:

Scaling-up centralized simulation. The technique proposed in CESIUM [2] is scaled up to thousands of distributed nodes by reducing the processing and memory requirements for node isolation. Moreover, it is scaled to execute off-the-shelf distributed applications written in modern Java while simulating key environment components, reproducing the concurrency, distribution, and performance characteristics of real-world deployments.

Providing meta-interfaces for observation and automation. Minha provides a programming interface to orchestrate large-scale virtual deployments of complete programs or standalone middleware layers with application stubs, aimed at being used within standardized testing frameworks. The same interface eases the collection of consistent snapshots and traces from distributed executions, suitable for visualization or evaluating global properties.

We evaluated Minha on a peer sampling service protocol and a large-scale key-value store. The results show that Minha is not only able to assess properties over a coherent, global snapshot of the system without imposing runtime overhead while at the same time allowing tests with a large number of nodes to run in cost effective way.

The rest of this paper is structured as follows. Section 2 outlines the state-of-the-art in simulation and emulation for testing distributed systems. Section 3 describes how the design and implementation of Minha provide a new trade-off between scale and observation completeness and scale. Sections 4 and 5 evaluate Minha. Section 6 concludes the paper.

2 Related Work

The ideal approach to test distributed systems software for race conditions is the use of implementation-level model checkers. Model checkers, such as MaceMC [18], Demeter [12], MoDist [39], dBug [35] and SAMC [22], intercept non-deterministic events of local and distributed programs (e.g. message arrivals, node crashes, and timeouts), and permute their ordering in systematic runs. This approach is very effective in discovering concurrency bugs, as it virtually explores the whole system state space. However, for large-scale applications, distributed model checking becomes unpractical and starts suffering from scalability issues due to state space explosion [22].

The next best approach is to run tests and then evaluate system-wide properties by logging the local state of each node independently along the execution and, periodically, send those logs to a centralized machine to be combined into a globally-consistent snapshot

[1] Minha is available as open source at `http://www.minha.pt`.

of the system that allows checking the desired predicates. For deployed systems, D^3S [27] proposes a simple language for writing distributed predicates that are checked on-the-fly at runtime. DCatch [26], in turn, aims to detect distributed concurrency bugs by resorting to a *happens-before (HB)* model. This work encodes the event causality into HB rules and builds a graph representing the timing relationships of several distributed concurrency and communication mechanisms. This approach has some drawbacks though. First, the execution details to be recorded at runtime must be defined *a priori*. Second, monitoring the nodes' local state is intrusive and induces both performance and space overhead, and finding the sweet spot between overhead and the necessary amount of information to be traced is far from trivial [27]. Third, setting up and running large scale tests requires a corresponding large scale distributed infrastructure to be available.

An important class of bugs introduced in the translation from algorithms to implementations, for instance, caused by the complexity of library operations and data structures used, can often be disclosed by running the system at large scale. Distributed systems researchers have been building platforms to address this challenge. EmuLab [16] provided a set of dedicated computer nodes and networking hardware that could be reconfigured to mimic different large scale systems. PlanetLab [32] uses a decentralized approach, therefore enabling a much larger and realistic platform to run off-of-the-shelf code. Unfortunately, experiments in PlanetLab are cumbersome to configure, deploy, and run. Splay [25] aims at easing the task of configuring the environment and running large-scale experiments, but limits the development to a specific framework and the Lua language.

The use of virtual machines and containers in public clouds, together with state-of-the-art orchestration software [11] makes it much easier for any developer to set up and run large tests. However, virtual nodes compete over the same physical resources, hence impacting negatively the performance of the system being evaluated and the accuracy of the measurements [34], possibly hiding the problems. Moreover, even if possible, it is still expensive to conduct extensive testing in public clouds.

Given the cost of running large scale tests and the difficulties in obtaining reliable, reproducible results, a number of proposals have focused on exploiting simulation to test actual implementations. Simulation is used extensively for distributed systems research, allowing simplified models to be tested in a very large scale [31], but they don't capture timeliness properties of implementation decisions and require code to be written in event-driven style, hence, with inversion of control. As an example, Neko [36] offers the ability to use simulation code as actual code, as long as its event-driven API is used in place of the standard Java classes.

An interesting trade-off is achieved by JiST [3] (Java in Simulation Time), a simulation kernel that allows event-driven simulation code to be written as Java threaded code, but avoids the overhead of a native thread for each simulated thread by using continuations. Unfortunately, this simulation kernel does not virtualize Java APIs and thus cannot be used to run most of the existing Java code. Moreover, it does not reflect the actual overhead of Java code in simulation time.

CESIUM [2] proposes the centralized simulation approach, in which the time effectively used to execute implementation code for each event is measured and reflected into simulation time. This is useful to detect issues such as the Cassandra failure detector bug [6], as it captures the timeliness properties of implementation code. This does not however address the general Java platform API and thus does not allow general code to be run. Moreover, by using Java class loaders for virtualization it imposes a large memory overhead and restricts simulations to a small number of nodes. UMLsim [1] is a similar proposal at the operating

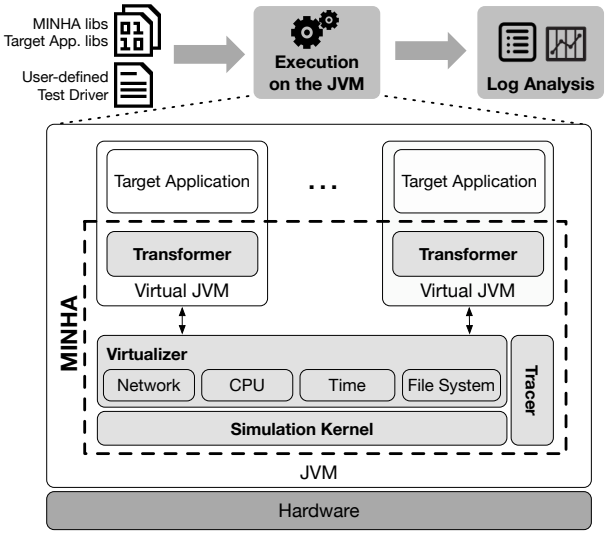


Figure 1 Execution flow and architecture of Minha.

system level, that virtualizes Linux while providing a simulated timeline and network. By requiring a full Linux installation for each node, it restricts attainable scale even more that CESIUM.

The ideal approach would thus have the ability to run unmodified implementation code, such as possible by using cloud computing, with the frugality and scalability of simulation, the ability to reproduce timeliness properties of CESIUM, and the ability to capture distributed snapshots of distributed debuggers. This the challenge addressed in this paper with Minha.

3 Minha Framework

Minha is a practical framework for testing large-scale distributed systems. Minha virtualizes multiple JVM instances within a single JVM and simulates key environment components, thus allowing reproducing the concurrency, distribution, and performance characteristics of an actual distributed system at a much lower resource cost.

The usage of Minha is shown at the top of Figure 1. From left to right, a *test driver* defines the execution scenario, such as the number of instances of the target application to be created and the global properties to be checked. The test driver is then executed, along with the target application and Minha libraries, on an off-the-shelf JVM. Properties can be checked in runtime and logs stored for further off-line checking and visualization.

At runtime, Minha acts as an interposition layer that takes control of the execution and steers it according to the testing scenario defined in the test driver. As show also in Figure 1, the main components of Minha are: the *simulation kernel*, the *virtualizer*, the *transformer*, and the *tracer*. Briefly, the transformer converts application and middleware code into an event-driven simulation that interacts with models provided by the virtualizer. Both run on an event-driven simulation kernel. The tracer collects information for off-line use. The remainder of this section describes how the design and implementation of Minha address the twin challenge of scale and observability of distributed systems software.

3.1 Achieving scale

Simulating multiple processes in the JVM requires isolating their code and preventing their executions from interfering with each other. Prior work typically addresses this issue by using a separate Java class loader for each process [2], that provides private copies of code and data for each virtual process. However, this approach severely limits attainable numerical scale, as each virtual process has to load, transform, compile, and store its own copy of each class.

A second aspect of scale is the size and complexity of the application and middleware that can be loaded. Current systems make use of large portions of the Java platform API in addition to the basic networking and time interfaces that have been intercepted in previous proposals [2]. Simply replacing all Java API with simulation models would lead to a very large development effort while impairing compatibility.

Minha's transformer addresses these challenges by using single class loader for all virtual processes and converts the original platform libraries to use simulation models of external resources, as happens for user provided middleware. As process isolation still needs to be enforced and the JVM forbids class loaders from rewriting native classes, Minha uses the ASM Java bytecode manipulation and analysis framework [5] to perform the following bytecode modifications:

Redirecting references to virtualized classes. Direct references to classes that are replaced by simulation models (e.g., `java.net.Socket`) are rewritten. Simulation models need then to be written, with same same interfaces, and containing simulation logic to reproduce their behaviors. This can however be done incrementally: As new applications demand new platform interfaces that haven't yet been modeled, they fail and report the problem. Experience shows that having implemented models for a moderate subset of the platform API supports many interesting distributed middleware components.

Redirecting static instance variables. This transformation moves static instance variables to regular instance variables in an auxiliary class. It then creates and uses static setter and getter methods for each of them. These methods use a map in the simulation model of each process to store and retrieve the correct instance, thus enforcing isolation.

Redirecting references to renamed classes and methods. Transformed platform classes are renamed to a separate package, thus circumventing the restriction to modifying them. Therefore, references to these classes and to classes that are replaced with simulated versions are re-directed to the new package by transforming their respective callers. Individual static methods in platform classes that do not need to be replaced as a whole (e.g., `System.currentTimeMillis()`) are simply redirected to simulated versions.

Moreover, a subset of classes, containing the simulation kernel and environment models, are kept *global* by delegating their load to the system's class loader. In contrast to isolation provided by using multiple class loaders and the Java security manager, the isolation between virtualized nodes in Minha is not designed for containing any malicious code or attack. This is however not needed, as Minha is used only for testing and all nodes are inherently trusted. This has the advantage of providing a controlled channel for virtual JVMs to interact with each other, that can easily be exploited by the user-defined test driver.

Finally, the last challenge to scale is in the discrete event simulation kernel. It keeps a list of future events per simulated timeline, scheduled to execute at target simulation instants. To scale up to large simulations, Minha's simulation kernel supports multiple timelines for parallel execution in multi-core systems. Timelines in a single simulation are

conservatively synchronized with a time window [28]: current time for events executing in parallel in different timelines differs by at most a constant W. This exploits the fact that Minha maps one or more simulated hosts to each timeline and that the network latency is at least W. Therefore, message delivery events scheduled on a different timeline are guaranteed to be properly ordered.

Parallel simulation in Minha is optimized to use one timeline for each available processor core. This is achieved by using a concurrent non-blocking data structure to store the event list and a non-blocking algorithm to keep track of the synchronization window. In detail, each thread tries to update the global lowest time of an event executing across all timelines, spinning until its next event handler can be safely executed. This approach works well in cases where events are evenly spread across all timelines, as typically happens when simulating large distributed systems in a few processor cores.

3.2 Achieving observability

Minha's programming interface is targeted at automated tests and scripting and combines reflection, to represent entities in the simulation domain, with an interface to inject events and interact with those entities.

To set up and control the simulation, the test driver API provides the following entities that describe and manipulate the simulated system: `World` represents the system as a whole, keeping track of global simulated time and allowing hosts to be created and enumerated. A `Host` describes a simulated host with processing and storage resources and attached to the network with an address. It allows creating and enumerating simulated processes. A `Process` keeps a private address space with its own copy of `static` variables and allows invocation of methods. Both hosts and processes can be terminated, to simulate crash faults.

Execution in the context of a simulated process is started by creating an `Entry` proxy for some interface and scheduling an invocation. Each entry point corresponds to a thread in the simulated process. Using the interface, it can specify arbitrary delays or an absolute schedule and if it is executed synchronously, implicitly running the simulation until the invocation returns, or asynchronously, providing a future to wait for and retrieve the result whenever the simulation has advanced enough. The simulation can also be run for predetermined periods of time, easing periodic observation. The API also provides the ability to exit the simulation to execute code that performs global observation or logging. This is achieved with an `Exit` proxy that ensures that the simulation is stopped on invocation and restarted on return. An example using the API is shown in Figure 3 and further discussed in Section 4.

The main challenge addressed is ensuring that only consistent global states can be inspected, regardless of the simulation containing multiple threads in a number of virtual processes. First, all blocking operations, such as synchronization and calls to the platform, are replaced with calls to simulated synchronization primitives by modifying the compiled bytecode at load time. Second, the transformed code executes consistently with simulation time. This ensures that: *i)* the application thread advances only in the context of an simulation event; *ii)* the execution time observed is reflected in the usage of a simulated CPU core; and *iii)* the waiting time (e.g., when reading from disk) is computed by the simulation and not by actual contention.

Figure 2 shows in detail how this is achieved. From the bottom to the top, the Control Thread (CT) originates in the Simulation Kernel and executes discrete Event Handling procedures. In contrast to common discrete event simulation practice, these do not directly modify model variables. Instead, they allow an Application Thread (AT) that executes test driver and application code at User Level to advance. An AT thus progresses as follows:

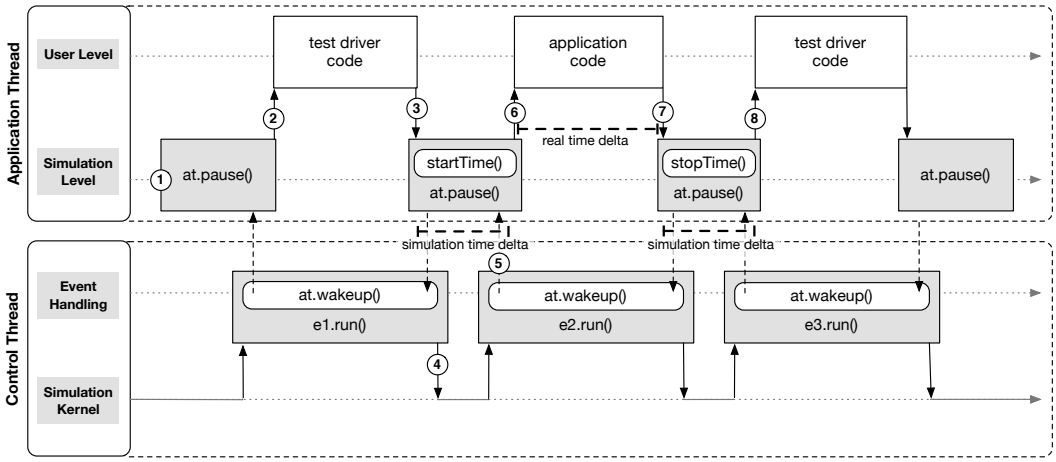


Figure 2 Example of how simulation time and execution time are coordinated in Minha.

(1) When started, the AT stopped using the `pause()` method to wait for its turn; when the corresponding event is scheduled by the Simulation Kernel, the CT signals the AT, that can then execute test driver code (2). When the test driver is done, (3) it starts accounting real time with `startTime()`. This pauses the AT and returns control to the Simulation Kernel, to wait for its turn (4). When simulation time has advanced, the AT is finally signaled to start executing application code (6). The time elapsed while executing application code is measured with the CPU cycle counter in `stopTime()` (7) and used to advance simulation time accordingly. This means that further events, either in application or test driver code (8) will be scheduled appropriately in simulation time.

This allows the passing of real time while executing code to influence simulation time, thus reproducing the performance characteristics as needed to disclose scaling bugs. However, since it is an event driven simulation, global state is consistent and can be observed by direct inspection while the simulation is stopped. Moreover, as logging is done in user-defined test driver code, outside the periods that measure real time, it has no impact in measured performance and does not disturb the system under test.

Note that the test driver thread could get blocked in synchronization primitives within the target application and middleware, leading to a deadlock. However, as synchronization primitives have been replaced by simulated counterparts, these will recognize that the invoking thread is the test driver and avoid blocking it. The developer writing the test driver has however to be careful to make sure that the code used for observation is not susceptible to inconsistent internal state.

In addition to directly checking properties, the tracer component allows logging events of interest at runtime for off-line processing. Currently, Minha is configured to trace events regarding: thread synchronization (i.e. *fork/join/start/end* events), inter-node communication (i.e. socket *send/receive* events), and *read/write* accesses to variables indicated by the programmer, if any. Recall that capturing memory accesses requires instructing the transformer to dynamically instrument the target application's bytecode with calls to the tracer, which may incur additional overhead during the execution. In contrast, all the remaining events are captured by Minha at the simulator side (i.e., outside the application), hence they do not impose any slowdown, as opposed to what happens in a real deployment.

An additional feature of the traces produced by Minha is that events are logged in a coherent global order, due to the framework's centralized and virtualized nature. This is particularly useful for debugging. In fact, Minha comes with a built-in diagram generator

that provides a visual representation of the execution according to the information stored in the trace file. Section 4.2 illustrates the benefits of this feature using the Cyclon example of Section 4.

The event trace is produced in JSON format, which can later be consumed by external tools. Events are stored as JSON objects, containing fields regarding: the thread identifier, the type of event (as indicated in tracer's description in Section 3), the timestamp, and, for inter-node communication events, a message unique identifier, and the source and destination node addresses. Minha's built-in visualizer consists of a Javascript module that uses the `SVG.js` library to generate a graphical representation of the event trace as a space-time diagram [20]. An example is shown in Figure 4 and discussed in the next section.

4 Use Case: Peer Sampling Service

In this section, we show how Minha can be used to test the properties of the overlays generated by the peer sampling service (PSS) Cyclon [37, 29]. A PSS is a mechanism, widely used by gossip-based peer-to-peer systems, that provides each node of the system with a partial *view* of the network. The overlay network used by the gossip protocol to spread information is thus defined by the logical graph that emerges from the union of all nodes' views at a given instant. Since the effectiveness and efficiency of the dissemination depends heavily on the properties of the overlay, the PSS refreshes each node's view from time to time to account for peer joins and leaves.

4.1 Test application

For the purpose of this example, we will focus on a particular PSS named Cyclon [37]. In a nutshell, the Cyclon protocol consists in a series of *shuffle* rounds, where pairs of neighbor nodes exchange a subset of randomly sampled peers from their views. The shuffle procedure works as follows. Each node assigns an *age* value to the node references in its view. This value is incremented by one at the beginning of a new exchange round, every T units of time. Upon starting a new shuffle, nodes pick the neighbor with highest age and send to it a random subset of other peers in their view, replacing the oldest node's reference with a self-reference of age 0. In turn, when a shuffle message is received, nodes first reply with a subset of peers randomly selected from their own view, and then update their views with the references received, replacing the entries previously sent. As a result of shuffle operations, nodes alive have their views refreshed and nodes that left the system are eventually removed from every view. This way, Cyclon is able to cope with dynamism.

The experimental results in Cyclon's original paper [37] shown that this protocol is able to generate network overlays with properties similar to random graphs, even starting from non-random topologies. This ability is particularly relevant for peer-to-peer systems that have to handle high churn, as random overlays exhibit low diameter and are able to maintain connectivity even in the presence of massive node failures.

The results presented in the original paper were obtained solely from simulations though, so it remains unclear how would the protocol behave in an actual distributed system. In such a real-world setting, the Cyclon protocol can be implemented by means of two execution threads: an *active thread* that is responsible for initiating a new shuffle with a neighbor, and a *passive thread* that operates as a message handler, which receives and processes the messages sent by other nodes. The portions of Cyclon executed by the two threads are detailed in Algorithms 1 and 2, respectively.

Algorithm 1 – **Cyclon Active Thread at Node p.**

```
Init:
V_size ← size of p's partial view
view ← p's view containing V_size references to other peers
S_size ← number of peers exchanged during a shuffling
         operation (S_size ≤ V_size)

for every T time units do
    // increase by one the age of all neighbors
    ageGlobal(view)
    // pick node q with highest age among all neighbors
    q = getOldestPeer(view)
    // select a random subset of S_size neighbors
    peers = selectPeers(view, S_size)
    // send list of peers to q, indicating that this message is a request for a shuffle
    send(q, {REQ, p, peers})
end for
```

Algorithm 2 – **Cyclon Passive Thread at Node p.**

```
while true do
    // receive a list of peers sent by node q
    {t, q, peersRcv} = receive()
    if t = REQ then
        // select a random subset of S_size neighbors
        peersSnd = selectPeers(view, S_size)
        // send list of peers to q, indicating that this message is a reply to the shuffle
        send(q, {REP, p, peersSnd})
    end if
    // incorporate list of peers received into its own view
    updateView(view, peersRcv)
end while
```

4.2 Test driver

To conduct the experiment in Minha, one can use a test driver like the one presented in Figure 3. The test driver starts by creating a new simulation scenario with 500 hosts, each corresponding to a Cyclon node, using Minha's API (lines 2-3). Nodes are then prepared for execution by enqueuing the method `run()` in the simulator (lines 6-8). This method is responsible for spawning Cyclon's passive and active threads, as well as initiating the node's view with references to its 11 subsequent neighbors.

The test driver proceeds with an instruction to let the simulation run for one second, thus allowing the Cyclon nodes to bootstrap (line 11). The core of the test run consists of 100 simulation cycles, in which Minha lets the application code of each instance execute for 5 seconds, before giving back the control to the test driver code (lines 15-21). At this point, the test driver consults the local state of each node, namely the composition of its view, and logs it into a trace file (lines 17-20). We highlight that these observations are performed transparently to the application and without incurring runtime overhead. Since all nodes are *paused*, the local views are obtained from a global snapshot of the system, thus allowing reenacting the actual overlay network at that exact moment.

4.3 Log analysis

Off-line log analysis is particularly well suited to discover message race conditions. As an example, we search logs obtained with Minha for cases in which a node is involved in more that one concurrent shuffle operations. This fact is not contemplated in the original Cyclon algorithm, nor in the simulations presented in the original paper [37].

```
//create a new simulation with 500 Cyclon peers
World world = new Simulation();
Entry<ICyclon>[] peers = world.createEntries(500, ICyclon.class, CyclonImpl.class.
    getName());

//start Cyclon peers
for (int i = 0; i < 500; i++){
    peers[i].queue().run();
}

//allow nodes to bootstrap
world.run(1,TimeUnit.SECONDS);

//let the application code execute for 5s
//and then observe the overlay
for(int i=0; i <= 100; i++) {
   world.run(5 , TimeUnit.SECONDS);
   for(ICyclon c : peers){
        //inspect node's view and store it into log
        logView(c.getView());
    }
}
```

Figure 3 Test driver for running Cyclon on Minha.

On the other hand, these atomicity violations can easily occur in real settings, and may hamper the properties of the overlay generated, especially in highly dynamic environments. Figure 4 illustrates a detail of a time-space diagram of the execution plotted from the event trace captured during the simulation for our experiment with Cyclon. To improve readability, we only depict the timelines of the two Cyclon threads (namely, the active thread and the passive thread) for the first four nodes. From the diagram, it can be observed that the active thread starts by spawning the passive thread and then proceeds to sending shuffle requests. In turn, the passive thread is responsible for receiving incoming messages and reply back to complete the view exchange, as defined by the protocol.

This time-space diagram confirms the interesting scenario: The atomicity of a shuffle operation is not guaranteed in practice (see the dashed box in Figure 4). In fact, a node can receive a new shuffle request from a third node in-between swapping views with a neighbor. Since the view updates performed by the passive thread upon receiving shuffle requests and replies are not commutative, it results in a view that is not anticipated in the algorithm. A possible solution to address this issue is to store incoming requests in a queue until the awaited reply arrives [17].

4.4 Scalability

This section evaluates how Minha's performance scales with the number of nodes simulated. In particular, we measured the execution time of Minha when varying both the number of nodes and the number of cycles considered in the simulation within the range $\{10, 100, 1000\}$. The experiments were performed on a machine with a 3.4 GHz Dual-Core Intel i3 Processor, 8 GB of RAM and a 7200 RPMs SATA disk. The results, averaged for three runs, are depicted in Figure 5.

As expected, the figure shows that Minha's execution time grows proportionally to the size of the simulation both in terms of number of Cyclon peers and number of cycles, making it efficient and practical for in-house testing of large-scale distributed systems as a

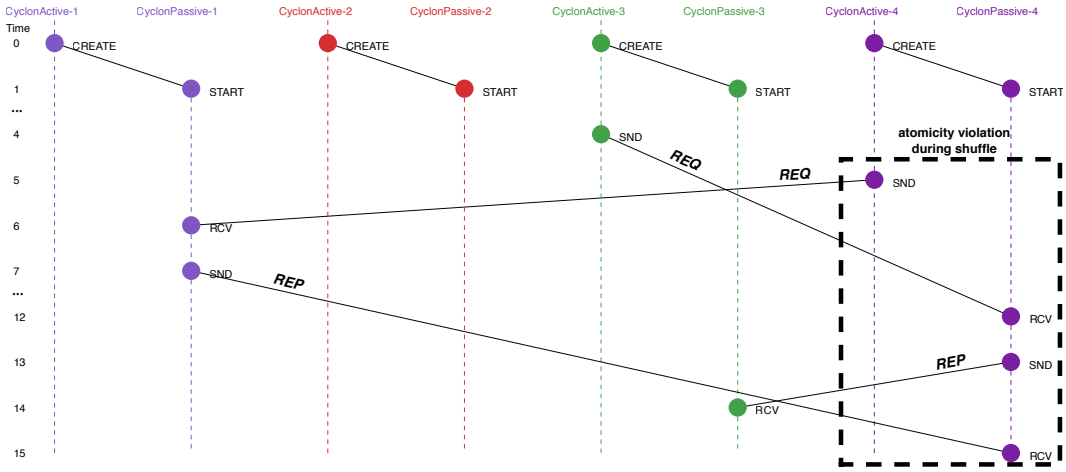


Figure 4 Space-time diagram generated by Minha's visualizer from the event trace captured during the simulation of Cyclon. The values on the left represent logical time ticks, whereas events *CREATE*, *START*, *SND*, *RCV*, denote, respectively, the creation and beginning of a thread, and the send and receive of a message. Messages labeled as *REQ* indicate shuffle requests and as *REP* indicate shuffle replies. The dashed box highlights the fact that atomicity is not guaranteed during a view exchange procedure, as new shuffle requests may arrive in the meantime.

simulation of the Cyclon protocol with 1000 nodes and 100 cycles (which suffice to assess most properties of the overlay, as shown in Section 4.2) takes only around 13 minutes with a minimal hardware configuration.

5 Use Case: Data Store

In this section, we provide a second example of Minha using the DataFlasks peer-to-peer database system.[2] This is used to show that Minha copes with an unmodified larger application that makes use of more features of the Java platform and external libraries. Moreover, we compare the scalability and cost of tests with the common alternatives of setting up a real distributed system or using virtual machines.

5.1 Global Property Checking

The DataFlasks distributed data store was designed to ensure availability of data in the presence of varying levels of node churn [30]. To assess this property, we ran DataFlasks on Minha with different levels of churn and wrote a test driver to compute the number of data replicas stored in the system at a given instant Churn is easily injected in using the test driver API by invoking `close()` on `Host` objects, to remove them from the system, or by creating new `Host` instances to bring them back. Computing the number of keys in the system requires inspecting the storage of each node to check the data it is holding and combining such information with that of the other nodes.

In the experiments, we instructed Minha to run the verification code and compute the average number of replicas that the system holds at execution intervals of 1 second. We plot the results of the experiments in Figure 6 for a simulation of 700 seconds. The results

[2] We used the version of DataFlasks publicly available at `github.com/fmaia/dataflasks`

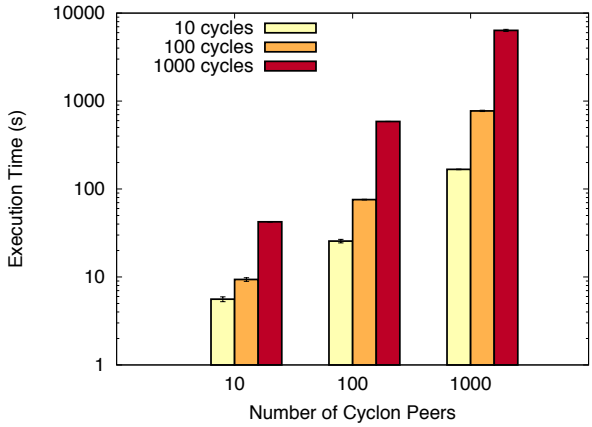


Figure 5 Minha's execution time (in log scale) for simulations considering different configurations of Cyclon.

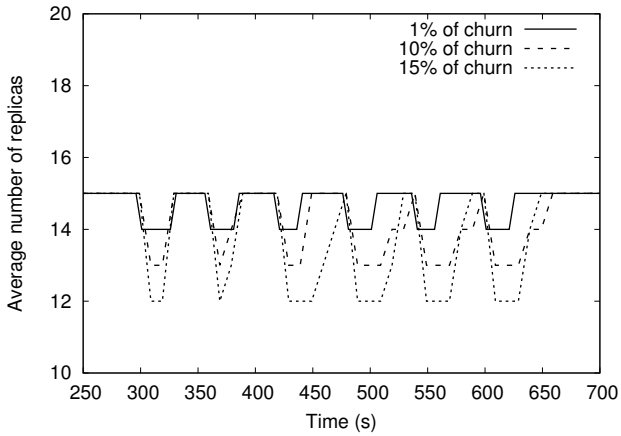


Figure 6 Minha's dynamic property checking applied to the evaluation of DataFlasks' replica maintenance properties.

show that, as expected, the average number of data replicas varies with churn (the higher the churn, the fewer replicas exist in the system), although DataFlasks is always able to eventually recover the keys lost. In fact, the system ends the experiment maintaining the expected mean number of data replicas. Checking the same property in a real-world deployment would require extensive logging and cumbersome synchronization mechanisms.

5.2 Performance and Resource Usage

For this experiment, we considered three different three different deployment configurations for DataFlasks:

Configuration 1 (Commodity). We considered a deployment built from a set of several commodity hosts. Each commodity host is randomly selected at startup time from a pool of resources equipped with either *i)* a 3.1 GHz Dual-Core Intel i3 Processor, 8 GB of RAM and a 7200 RPMs SATA disk, *ii)* a 3.4 GHz Dual-Core Intel i3 Processor, 8 GB of RAM and a 7200 RPMs SATA disk, or *iii)* a 3.7 GHz Dual-Core Intel i3 Processor, 8 GB of RAM and

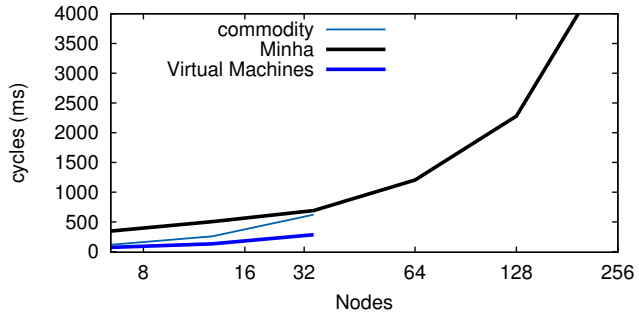


Figure 7 Time to execute the workload in the DataFlasks key-value store for the three deployment configurations.

a SSD disk. All hosts are interconnected through a switched Gigabit Ethernet. The total number of commodity hosts available at deployment time was limited to 36, which aims to represent a scenario where the available resources for testing are not comparable with the equivalent to those expected in a production deployment.

Configuration 2 (Virtual Machines). We considered a deployment built from a single server grade machine equipped with an 2.3 GHz Intel Xeon E5-2670 v3 Processor, 94GB of RAM and a 7200 RPMs SATA disk. This machine was configured to deploy a set of 32 virtual machines interconnected through a virtualized switched Ethernet.

Configuration 3 (Minha). We considered a single commodity host, extracted from the pool of resources described in the first scenario, on which we deployed Minha.

For each one of the previous configurations, we deployed DataFlasks with an increasing number of nodes within the range $\{8, 16, 32, 64, 128, 256\}$. We then performed the experiments by running a simple write workload on top of DataFlasks using YCSB [9], in a total of 5 independent runs for each configuration. For each experiment, we measured the time required to replicate data across all the necessary replicas and used it to compare the different configurations. Figure 7 depicts the experimental results.

Figure 7 shows that besides scaling to larger system sizes than a configuration with multiple virtual machines, Minha is able to do it on a single commodity host that represents 1/36 of the cost of a real deployment depicted in either the first or second configurations. These results support the claim that Minha is able to accurately simulate large-scale distributed systems with much less resources than traditional approaches.

6 Conclusions and Future Work

Distributed systems are notoriously hard to test, mainly due to their large state space and the difficulty in deploying large infrastructures. This paper addresses the challenges of scale and observability in Minha, a simulation framework aimed at easing the burden of testing large-scale distributed systems. Minha virtualizes multiple JVM instances within a single JVM, while simulating key environment components, thus reproducing the concurrency, distribution, and performance characteristics of an actual system. Minha also helps assessing the application's correctness by allowing checking distributed properties on globally-consistent snapshots of the system. Moreover, due to time virtualization, these system-wide assertions can be performed transparently to the target application and without affecting its runtime performance.

Our experiments with a large-scale key-value store and a peer sampling service show that Minha can accurately reproduce the characteristics of real-world deployments with fewer resources than traditional approaches, and is effective in assessing system-wide properties.

We believe that Minha opens a number of interesting research opportunities in the field of distributed systems testing and debugging. For example, Minha can be extended with model checking capabilities to systematically explore the execution space of distributed systems and discover new bugs. Furthermore, Minha's event traces, which are totally ordered and logged without any runtime overhead for the target application, can be combined with bug detection techniques to also detect problems automatically.

References

1. Werner Almesberger. umlsim-A UML-based simulator. In *10th International Linux System Technology Conference (Linux-Kongress 2003)*, pages 202–213, 2003.
2. G. A. Alvarez and F. Cristian. Applying simulation to the design and performance evaluation of fault-tolerant systems. In *SRDS '97*, pages 35–42, October 1997.
3. Rimon Barr, Zygmunt J Haas, and Robbert van Renesse. JiST: An efficient approach to simulation using virtual machines. *Software: Practice and Experience*, 35(6):539–576, 2005.
4. Y Bertot and P Castéran. Interactive Theorem Proving and Program Development—Coq'Art: The Calculus of Inductive Constructions (2004).
5. Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.
6. Bug CASSANDRA-6127: vnodes don't scale to hundreds of nodes. `https://issues.apache.org/jira/browse/CASSANDRA-6127`.
7. Bug ZOOKEEPER-2212: distributed race condition related to QV version. `https://issues.apache.org/jira/browse/ZOOKEEPER-2212`.
8. Apache Cassandra. `http://cassandra.apache.org`.
9. Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC '10*, pages 143–154, New York, NY, USA, 2010. ACM.
10. Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. An Empirical Study on the Correctness of Formally Verified Distributed Systems. In *EuroSys '17*, New York, NY, USA, 2017. ACM.
11. F. Gortázar, M. Gallego, M. Donato, E. Pages, A. Edmonds, G. Tuñón, A. Bertolino, G. De Angelis, A. Cervantes, T. Bohnert, A. Willner, and V. Gowtham. The ElasTest Platform: Supporting Automation of End-to-End Testing of Large Complex Applications. ElasTest project whitepaper, November 2018. URL: `https://elastest.io/resources/ElasTest_white_paper.pdf`.
12. Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical software model checking via dynamic interface reduction. In *SOSP '11*, pages 265–278. ACM, 2011.
13. Diwaker Gupta, Kashi Venkatesh Vishwanath, Marvin McNett, Amin Vahdat, Ken Yocum, Alex Snoeren, and Geoffrey M. Voelker. DieCast: Testing Distributed Systems with an Accurate Scale Model. *ACM Trans. Comput. Syst.*, 29(2):4:1–4:48, May 2011.
14. Apache Hadoop. `http://hadoop.apache.org`.
15. Naohiro Hayashibara, Xavier Défago, Rami Yared, and Takuya Katayama. The Phi Accrual Failure Detector. In *SRDS*. IEEE Computer Society, 2004.
16. Mike Hibler, Robert Ricci, Leigh Stoller, Jonathon Duerig, Shashi Guruprasad, Tim Stack, Kirk Webb, and Jay Lepreau. Large-scale Virtualization in the Emulab Network Testbed. In *USENIX 2008 Annual Technical Conference*, ATC'08, 2008.

17 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.

18 Charles Killian, James W Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *NSDI '07*, 2007.

19 Ignacio Laguna, Dong H. Ahn, Bronis R. de Supinski, Todd Gamblin, Gregory L. Lee, Martin Schulz, Saurabh Bagchi, Milind Kulkarni, Bowen Zhou, Zhezhe Chen, and Feng Qin. Debugging High-performance Computing Applications at Massive Scales. *Commun. ACM*, 58(9):72–81, August 2015.

20 Leslie Lamport. Time clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, July 1978.

21 Leslie Lamport. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

22 Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. SAMC: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *OSDI '14*. USENIX Association, 2014.

23 Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *ASPLOS '16*. ACM, 2016.

24 Tanakorn Leesatapornwongsa, Cesar A. Stuardo, Riza O. Suminto, Huan Ke, Jeffrey F. Lukman, and Haryadi S. Gunawi. Scalability Bugs: When 100-Node Testing is Not Enough. In *HotOS '17*, pages 24–29. ACM, 2017.

25 Lorenzo Leonini, Étienne Rivière, and Pascal Felber. SPLAY: Distributed Systems Evaluation Made Simple (or How to Turn Ideas into Live Systems in a Breeze). In *NSDI*, volume 9, pages 185–198, 2009.

26 Haopeng Liu, Guangpu Li, Jeffrey F. Lukman, Jiaxin Li, Shan Lu, Haryadi S. Gunawi, and Chen Tian. DCatch: Automatically detecting distributed concurrency bugs in cloud systems. In *ASPLOS '17*. ACM, 2017.

27 Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. D3S: Debugging Deployed Distributed Systems. In *NSDI '08*. USENIX Association, 2008.

28 B. D. Lubachevsky. Efficient Distributed Event-driven Simulations of Multiple-loop Networks. *Commun. ACM*, 32(1):111–123, January 1989.

29 Nuno Machado, Francisco Maia, Miguel Matos, and Rui Oliveira. BuzzPSS: A Dependable and Adaptive Peer Sampling Service. In *LADC '16*. IEEE Computer Society, 2016.

30 Francisco Maia, Miguel Matos, Ricardo Vilaça, José Pereira, Rui Oliveira, and Etienne Rivière. DATAFLASKS: Epidemic Store for Massive Scale Systems. In *SRDS '14*. IEEE Computer Society, 2014.

31 A. Montresor and M. Jelasity. PeerSim: A scalable P2P simulator. In *International Conference on Peer-to-Peer Computing*, 2009.

32 Larry Peterson and Timothy Roscoe. The design principles of PlanetLab. *ACM SIGOPS operating systems review*, 40(1):11–16, 2006.

33 Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Middleware '01*, London, UK, UK, 2001. Springer-Verlag.

34 Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. *Proc. VLDB Endow.*, 3(1-2):460–471, September 2010. doi:10.14778/1920841.1920902.

35 Jiri Simsa, Randy Bryant, and Garth A Gibson. dBug: Systematic Evaluation of Distributed Systems. In *SSV '10*, 2010.

36 Peter Urban, Xavier Défago, and André Schiper. Neko: A single environment to simulate and prototype distributed algorithms. In *Information Networking, 2001. Proceedings. 15th International Conference on*, pages 503–511. IEEE, 2001.

37 Spyros Voulgaris, Daniela Gavidia, and Maarten van Steen. CYCLON: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13(2):197–217, 2005.

38 Yang Wang, Manos Kapritsos, Lara Schmidt, Lorenzo Alvisi, and Mike Dahlin. Exalt: Empowering Researchers to Evaluate Large-scale Storage Systems. In *NSDI'14*, pages 129–141, Berkeley, CA, USA, 2014. USENIX Association.

39 Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent model checking of unmodified distributed systems. In *NSDI '09*. USENIX Association, 2009.

Fast Lean Erasure-Coded Atomic Memory Object

Kishori M. Konwar
Department of EECS, MIT, Cambridge, USA
kishori@mit.edu

N. Prakash [1]
Intel Inc, OR, USA
prakashnarayanamoorthy@gmail.com

Muriel Médard
Department of EECS, MIT, Cambridge, USA
medard@mit.edu

Nancy Lynch
Department of EECS, MIT, Cambridge, USA
lynch@csail.mit.edu

Abstract

In this work, we propose FLECKS, an algorithm which implements atomic memory objects in a multi-writer multi-reader (MWMR) setting in asynchronous networks and server failures. FLECKS substantially reduces storage and communication costs over its replication-based counterparts by employing erasure-codes. FLECKS outperforms the previously proposed algorithms in terms of the metrics that to deliver good performance such as storage cost per object, communication cost a high fault-tolerance of clients and servers, guaranteed liveness of operation, and a given number of communication rounds per operation, etc. We provide proofs for liveness and atomicity properties of FLECKS and derive worst-case latency bounds for the operations. We implemented and deployed FLECKS in cloud-based clusters and demonstrate that FLECKS has substantially lower storage and bandwidth costs, and significantly lower latency of operations than the replication-based mechanisms.

2012 ACM Subject Classification Theory of computation → Distributed computing models

Keywords and phrases Atomicity, Distributed Storage System, Erasure-codes

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.12

1 Introduction

In the recent years, the demand for efficient and reliable large-scale distributed storage systems (DSSs) has grown at an unprecedented scale. DSSs that store massive data sets across several hundreds of servers are commonly used for both industrial and scientific applications, and numerous Internet-based applications. Many applications demand concurrent and consistent access to the stored data by multiple writers and readers. Therefore, some form of consistency must be guaranteed of the stored objects is essential for the application developer to reason about the correctness of the application. The consistency model we adopt is *atomicity*, also often referred to as *strong consistency*. Atomic consistency gives the users of the data service the impression that the various concurrent read and write operations happen sequentially. Therefore, strong consistency or linearizability is the most preferred form of consistency guarantee. However, providing strong consistency is a non-trivial task in most practical distributed storage systems due the asynchronous behavior of the communication and component failures endemic in any large network. Also, the ability to withstand failures and network delays are essential features of any robust DSS. The traditional solution for

[1] This work was done while the author was still at MIT.

© Kishori M. Konwar, N. Prakash , Muriel Médard, and Nancy Lynch;
licensed under Creative Commons License CC-BY
23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller; Article No. 12; pp. 12:1–12:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

emulating an atomic fault-tolerant shared storage system involves replication of data across the servers. Perhaps, the earliest of replication-based algorithms atomic memory emulation in asynchronous networks appear in the work by Attiya, Bar-Noy and Dolev [4] (we refer to this as the ABD algorithm).Replication based strategies incur high storage costs; for example, to store a *value* (an abstraction of a data file) of size 1 MB across a 5-server system, the ABD algorithm replicates the value in all the 5 servers, which blows up the worst-case *storage cost* to 5 MB. Additionally, every write or read operation has a worst-case *communication cost* of 5 MB. The communication cost, or simply the cost, associated with a read or write operation is the amount of total data in bytes that gets transmitted in the various messages sent as part of the operation. Since the focus in this paper is on large data objects, the storage and communication costs include only the total sizes of stable storage and messages dedicated to the data itself. Ephemeral storage and the cost of control communication is assumed to be negligible. Under this assumption, we further *normalize* both the storage and communication costs with respect to the size of the value, say v, that is written, i.e., we simply assume that the size of v is 1 unit (instead of 1 MB), and say that the worst-case storage or read or write cost of the ABD algorithm is n units, for a system consisting of n servers.

Erasure codes provide an alternative way to emulate fault-tolerant shared atomic storage, with the added benefit of reducing storage cost. In comparison with replication, algorithms based on erasure codes significantly reduce both the storage and communication costs of the implementation. An $[n, k]$ erasure code splits the value v of size 1 unit into k elements, each of size $\frac{1}{k}$ units, creates n *coded elements*, and stores one coded element per server. The size of each coded element is also $\frac{1}{k}$ units, and thus the total storage cost across the n servers is $\frac{n}{k}$ units. For example, if we use an $[n = 5, k = 3]$ MDS code, the storage cost is simply 1.67 per unit of data, instead of 5 as in the case of replication-based algorithms, such as ABD. A class of erasure codes known as Maximum Distance Separable (MDS) codes have the property that value v can be reconstructed from any k out of these n coded elements. In systems that are centralized and synchronous, the parameter k is simply chosen as $n - f$, where f denotes the number of server crash failures that need to be tolerated. In this case, the read cost, write cost and total storage cost can all be simultaneously optimized. The usage of MDS codes to emulate atomic shared storage in decentralized, asynchronous settings is way more challenging, and often results in additional communication or storage costs for a given level of fault tolerance, when compared to the synchronous setting. Even then, as has been shown in the past [6, 10], significant gains over replication-based strategies can still be achieved while using erasure codes. The works in [6, 10] contain algorithms based on MDS codes for emulating fault-tolerant shared atomic storage, and offer different trade-offs between storage and communication costs.

The performance of a DSS that stores millions of objects, and accessed concurrently by hundreds of thousands of clients must excel in terms of several performance measures. While designing FLECKS algorithm we focused on the following key performance metrics that are often used by the systems researchers to evaluate the performance of such system. (*i*) *Storage cost* is the total number of bytes stored across all servers, must be low, which essentially increases the capacity of the storage system, and also reduces the cost of storing data for the user. (*ii*) *Maximum number of server failures* the system can experience without service interruption directly contributed to increases in data durability. (*iii*) *Number of rounds per operation* reduces the latency of operations, thereby increasing the throughput of clients' operations and also reduces overall messaging in the network. (*iv*) *Read cost* is the amount of data transmitted in order to complete a read operation. In most practical systems reads are several orders of magnitude more frequent than writes. Therefore, read cost, must be as

low as possible. (v) *Write cost* is the number of bytes transmitted during a write operation should be as low as possible, which would reduce latency of write and network bandwidth consumption.

Our Contributions. In this work, we present FLECKS, an erasure-code based, fault-tolerant algorithm for implementing MWMR atomic memory in asynchronous networks, with optimized storage and communication costs. When compared to other erasure-code based or replication-based atomic memory emulation algorithms, FLECKS achieves superior or comparable values for the performance metrics mentioned above. Moreover, FLECKS is the only such algorithm that scores reasonable values across all of the performance metrics (see Table 1), making it suitable for implementations in practical systems. Firstly, the storage cost of FLECKS is $(1+\delta)\frac{n}{k}$, where δ is the maximum number of writes concurrent with any read. In a typical DSS, the frequency of reads is 10,000+ fold more than that of writes [8]. Therefore, δ is rarely larger than 1 as reported in [7]. FLECKS exploits this to provide one-round reads, but occasionally, in the presence of concurrent writes, carries out a second round. This results in lower latency for most reads and increases throughput of the system. Writes always take two rounds. We would like to emphasize that δ is not explicitly hard-coded in FLECKS; therefore, is a run-time property. The underpinning idea behind FLECKS achieving lower storage cost is to use writes help garbage collect stale values, i.e., values introduced by previous writes. As a result, during the course of an execution, the additional storage cost due to the temporary increase of δ for individual object is small and transient. In a system with several hundred or more stored objects, the fraction of reads that experiences concurrent writes would be tiny (see third plot in Fig. 1). Therefore, when considered system wide, FLECKS achieves storage cost very close to the optimal value $\frac{n}{k}$ (discussed later in the context of Fig. 2 (a)). FLECKS can tolerate a maximum of $n-k$ server crashes, which is the maximum number of erasures tolerated by and MDS $[n,k]$ code. The read and write-communication costs are very comparable to the synchronous EC-based scenarios (see Table 1). We provide analytical proofs of *atomicity* and *liveness* properties of FLECKS. We also derived bounds for the read and write latency based on maximum message delay of Δ for any point-to-point message in the network. Finally, we implemented FLECKS, deployed our implementation, and ran experiments where our implementation can emulate a large number of atomic objects. We compare our results with an optimized replication-based algorithm adapted from ABD where we emulate a shared storage of up to 10,000 objects of various sizes. Our results corroborate our design goals and theoretical results on storage and communication cost bounds, and lower latency of reads and writes in FLECKS. For example, Fig. 1 shows that FLECKS (EC) has much lower latency, compared to the replication-based method (REP) for the read and write operations. Furthermore, it shows that most of the reads (GET) comprise of a single-round.

1.1 Comparison with Other Algorithms, and Related Work

There is a rich history of erasure coding based shared memory emulation algorithms [5, 6, 10–12, 14, 18]. In Table 1, we provide a comparison between FLECKS and other atomic memory algorithms. We add ABD as a benchmark to compare the performance metrics of the erasure-coded algorithms with replication based schemes. In [6], the authors provide two algorithms - CAS and CASGC - based on $[n,k]$ MDS codes, and these are primarily motivated with a goal of reducing the communication costs. Both algorithms tolerate up to $f=\frac{n-k}{2}$ server crashes, and incur a communication cost (per read or write) of $\frac{n}{n-2f}$. The CAS algorithm is a precursor to CASGC, and its storage cost is not optimized. In CASGC,

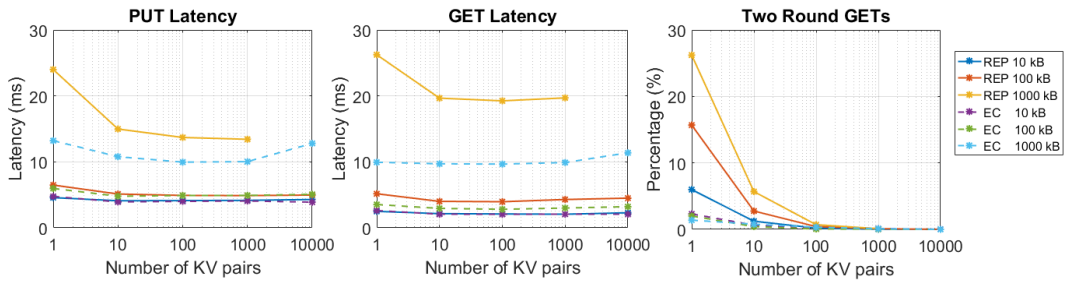


Figure 1 READ (GET) and WRITE (PUT) latencies, and percentage of READs with 2 phases for the multi object experiment. For each operation, a client accesses a object chosen uniformly at random. We compare $[n = 5, k = 3]$ FLECKS (EC) against 5-way replication (REP), for objects of sizes 10KB, 100KB and 1MB.

each server stores coded elements (of size $\frac{1}{k}$) for up to $\delta + 1$ different versions of the value v, where δ is a hard-coded upper bound on the number of writes that are concurrent with a read A garbage collection mechanism, which removes all the older versions, is used to reduce the storage cost. The worst-case total storage cost of CASGC is shown to be $\frac{n}{n-2f}(\delta + 1)$. Liveness and atomicity of CASGC are proved under the assumption that the number of writes concurrent with a read never exceeds δ. On the other hand, SODA [14] is designed to optimize the storage cost rather than communication cost, where a write cost is very high (n^2). In SODA, the parameter δ_w, which indicates the number of writes concurrent with a read, to bound the read cost. However, neither liveness or atomicity of SODA depends on the knowledge of δ_w; the parameter appears only in the analysis and not in the algorithm. But the effect of the parameter δ in CASGC is rather *rigid*. In CASGC, any time after $\delta + 1$ successful writes occurs during an execution, the total storage cost remains fixed at $\frac{n}{n-2f}(\delta+1)$, irrespective of the actual number of concurrent writes during a read. For a given $[n, k]$ MDS code, CASGC tolerates only up to $f = \frac{n-k}{2}$ failures, whereas SODA tolerates up to $f = n - k$ failures.

In [10], the authors present the ORCAS-A and ORCAS-B algorithms for asynchronous crash-recovery models. In this model, a server is allowed to undergo a temporary failure such that when it returns to normal operation, contents of temporary storage (like memory) are lost while those of permanent storage are not. Only the contents of permanent storage count towards the total storage cost. Furthermore they do not assume reliable point-to-point channels. The ORCAS-A algorithm offers better storage cost than ORCAS-B when the number of concurrent writers is small. Like SODA, in ORCAS-B coded elements corresponding to multiple versions are sent by a writer to reader, until the read completes. However, unlike in SODA, a failed reader might cause servers to keep sending coded elements indefinitely. RADON [15], an erasure-code based atomic memory algorithm which allows servers restarts, provides liveness guarantees under most practical network settings and allows efficient repair of crashed nodes. ARES [18] improves on the number of rounds compared to the previously known erasure-code based algorithms. From Table 1 is it evident that FLECKS strikes a balance among all the erasure-code based algorithms performs in all of the measures of performance.

1.2 Other related works

In [19], the authors consider algorithms that use erasure codes for emulating *regular* registers. Regularity [16] is a weaker consistency notion than atomicity. Applications of erasure codes to Byzantine fault tolerant DSS are discussed in [5, 12].

During the last few years several erasure-code-based DSS with strongly consistent distributed storage have become available. Cocytus [20] is an in-memory key-value store that guarantees strong consistency and reduces storage cost using erasure codes. The values are erasure coded and the coded elements are stored among a subset or group of servers, referred to as coding group, from the set of available servers.

Table 1 Performance metrics of replication-based, FLECKS and other algorithms with erasure-codes (for MDS code of dimension $[n, k]$) for atomic read/write memory emulation. δ is the maximum number of concurrent writes with any read during the course of an execution of the algorithm. In practice, $\delta < 4$ [7]. The optimal case is the use of EC in a synchronous system.

algorithm	max failures	rounds/ write	rounds/ read	repl or EC	storage cost	read cost	write cost	explicit δ?
ABD [4]	$\lfloor \frac{n-1}{2} \rfloor$	2	2	Repl.	n	$2n$	n	-
CASGC [6]	$\lfloor \frac{n-k}{2} \rfloor$	3	2	EC	$(\delta+1)\frac{n}{k}$	$\frac{n}{k}$	$\frac{n}{k}$	Yes
SODA [14]	$n-k$	2	2	EC	$\frac{n}{k}$	$(\delta+1)\frac{n}{k}$	$\frac{n^2}{k}$	No
ORCAS-A [10]	$\lfloor \frac{n-k}{2} \rfloor$	3	≥ 2	EC	n	n	n	Yes
ORCAS-B [10]	$\lfloor \frac{n-k}{2} \rfloor$	3	3	EC	∞	∞	∞	-
RADON$_c$ [15]	$\lfloor \frac{n-k}{2} \rfloor$	2	2	EC	$(\delta+1)\frac{n}{k}$	$(\delta+2)\frac{n}{k}$	$\frac{n}{k}$	Yes
ARES [18]	$\lfloor \frac{n-k}{2} \rfloor$	2	2	EC	$(\delta+1)\frac{n}{k}$	$(\delta+1)\frac{n}{k}$	$\frac{n}{k}$	Yes
FLECKS	$n-k$	2	≤ 2	EC	$(\delta+1)\frac{n}{k}$	$(\delta+1)\frac{n}{k}$	$\frac{n}{k}$	No
SYNCH EC	$n-k$	1	1	EC	$\frac{n}{k}$	1	$\frac{n}{k}$	-

Giza [7] is a recently proposed strongly-consistent multi-version object store and heavily used in Microsoft's OneDrive storage system. Giza is designed for cross-data center (cross-DC) object storage, which is deployed over 11 data-centers around the world. Giza uses FastPaxos [17] which, in the absence of concurrent writes, completes in one round trip, but in the case of concurrent updates, uses the more expensive consensus algorithm Paxos.

Recently, a large class of new erasure codes have been proposed and employed (see [9] for a survey) in DSS where the focus is on the efficient storage of immutable (like archival) data. Recovery of contents in failed servers is an important operation in such systems. These new codes offer the dual benefits of reduced storage cost as well as reduced repair cost during recovery from server failures. It remains to be seen whether the advantages of these codes carry over to systems that have consistency/concurrency requirements.

Document Structure. In Section 2, we provide the models and definitions. In Section 3 we describe FLECKS. Section 4 provides the proof for correctness and liveness guarantees for FLECKS along with bounds for storage and communication costs, and latency analysis of the operations. In Section 5, we discuss the implementation and experimental validation of FLECKS. Finally, in Section 6 we conclude our paper. Due to lack of space some of the proofs are omitted.

2 Model and Definitions

A shared atomic storage can be emulated by composing individual atomic objects. Therefore, we aim to implement a single atomic read/write memory object. Each data object takes a value from a set $\mathcal{V}$. We assume a system consisting of three distinct sets of processes: a set $\mathcal{W}$ of writers, a set $\mathcal{R}$ of readers and $\mathcal{S}$, a set of servers. Servers host data elements (replicas or encoded data fragments). Each writer is allowed to WRITE the value of a shared object, and each reader is allowed to READ the value of that object. Processes communicate via *messages* through *asynchronous*, *reliable* channels.

Executions. An *execution* of an algorithm A is an alternating sequence of states and actions of A starting with the initial state and ending in a state. An execution ξ is *well-formed* if each client does not invoke a one operation until it completed the previously invoked operation and it is *fair* if enabled actions perform a step infinitely often. In the rest of the paper we consider executions that are fair and well-formed. When process p *crashes* it stops executing any further step.

Write and Read Operations. An implementation of a read or a write operation contains an *invocation* action and a *response* action (such as a return from the procedure). An operation π is *complete* in an execution, if it contains both the invocation and the *matching* response actions for π; otherwise π is *incomplete*. We say that an operation π *precedes* an operation π' in an execution ξ, denoted by $\pi \rightarrow \pi'$, if the response step of π appears before the invocation step of π' in ξ. Two operations are *concurrent* if neither precedes the other.

Erasure Codes. ***Background on Erasure coding:*** In $FLECKS$, we use an $[n, k]$ linear MDS code [13] over a finite field $\mathbb{F}_q$ to encode and store the value v among the n servers. An $[n, k]$ MDS code has the property that any k out of the n coded elements can be used to recover (decode) the value v. For encoding, v is divided[2] into k elements $v_1, v_2, \ldots v_k$ with each element having size $\frac{1}{k}$ (assuming size of v is 1). The encoder Φ takes the k elements as input and produces n coded elements $c_1, c_2, \ldots, c_n$ as output, i.e., $[c_1, \ldots, c_n] = \Phi([v_1, \ldots, v_k])$. For ease of notation, we simply write $\Phi(v)$ to mean $[c_1, \ldots, c_n]$. The vector $[c_1, \ldots, c_n]$ is referred to as the codeword corresponding to the value v. Each coded element c_i also has size $\frac{1}{k}$. In our scheme we store one coded element per server. Without loss of generality, we associate the coded element c_i with server i, $1 \leq i \leq n$.

Liveness of operations. We require algorithms to satisfy certain liveness properties, specifically, in every fair execution that satisfies certain restrictions in terms of the number of failed nodes, we require every operation by a non-faulty client completes, irrespective of the behavior of other clients.

Storage and Communication Costs. We define the total storage cost as the size of the data stored across all servers, at any point during the execution of the algorithm. The communication cost associated with a read or write operation is the size of the total data that gets transmitted in the messages sent as part of the operation. We assume that metadata, such as version number, process ID, etc. used by various operations is of negligible size, and therefore, ignore this in the calculation of storage and communication cost. Further, we normalize both the costs with respect to the size of the value v; in other words, we compute the costs under the assumption that v has size 1 unit.

3 The FLECKS algorithm

The FLECKS algorithm is presented in three parts in Pseudocodes. 1, 2 and 3, corresponding to a writer, reader and server, respectively. The erasure-code parameter k is chosen as $k = n - f$, where f is the desired server-fault tolerance. By assumption, $f < n/2$, and thus

[2] In practice v is a file, which is divided into many stripes based on the choice of the code, various stripes are individually encoded and stacked against each other. We omit details of represent-ability of v by a sequence of symbols of $\mathbb{F}_q$, and the mechanism of data striping, since these are fairly standard in the coding theory literature.

Algorithm 1 Writer protocol in FLECKS: WRITE(v) at writer w.

```
Variables:
opnum, indicates the operation number for the writer. Initially 1.
put-data:
  Compute coded elements c_1, ..., c_n from v
  Send (opnum, c_i) to server s_i, 1 ≤ i ≤ n.
  Wait for responses from k servers. Let the responses be {z_i, 1 ≤ i ≤ k}.
  Compute z = max_i z_i

put-tag:
  Let t = (w, z)
  Send (t, opnum) to server s_i, 1 ≤ i ≤ n.
  opnum++. Terminate after receiving k acknowledgments.
```

Algorithm 2 Reader protocol in FLECKS: READ at reader r.

```
get-tag-data:
  Request final tuple from all servers
  Wait for responses from k servers.

  if all k responses have common tag then
    decode the corresponding value
    return the value.
  else
    compute the maximum received tag and call it t_req
    Let opnum_req be the corresponding opnum.
    collect all coded elements corresponding to t_req in list D_L

get-data:
  Send (t_req, opnum_req) to all servers.
  Collect every response (t, opnum, c) into D_L.
  for received tuple (t, opnum, c), do
    if ∃ k coded elements t in D_L then.
      decode the value v for tag t
      send read-complete to all servers
      return v
    else
      if t > t_req then
        send commit-tag(t, opnum) to all servers,
        Continue to wait for more tuples.
```

we get that $k > n/2$. The algorithm relies on the notion of quorums during both phases of the WRITE operation, and the first phase of the READ operation. The parameter k denotes the size of quorum in these phases, and is at least a majority since $k > n/2$.

Tags are used for version control of key values. A tag t is defined as a pair (z, w), where z is an positive integer and $w \in \mathcal{W}$ denotes the writer ID. We use $\mathcal{T}$ to denote the set of all possible tags. For any two tags $t_1, t_2 \in \mathcal{T}$ we say $t_2 > t_1$ if (i) $t_2.z > t_1.z$ or (ii) $t_2.z = t_1.z$ and $t_2.w > t_1.w$. The relation $>$ imposes a total order on $\mathcal{T}$.

Server-side Local Variables: Each server maintains the following local variables: $a)$ a List $L \subset Tags \times \mathbb{N} \times$ coded elements $\times \{Pre, Fin\}$, which forms a temporary storage for tag and coded-elements pairs during WRITE operations. The second entry indicates the operation number (opnum) of the writer whose entry is stored. The last entry's meaning will be described further in the text. $b)$ A finalized tuple $(t_f, opnum_f, c_{i,f})$. We refer to t_f as the finalized tag, $opnum_f$ as the finalized opnum, and $c_{i,f}$ as the finalized coded-element, $c)$ $Op(w), w \in \mathcal{W}$, indicating the last opnum received from writer w, and $d)$ the set $\mathcal{R}$ of outstanding READ requests. An element of $\mathcal{R}$ is the form $(r, t_{req}, opnum_{req})$.

We now describe the WRITE and READ operations with the help of Pseudocode 1, 2 and 3, and a high-level schematic diagram for the read and write operations are given in Fig. 2.

Algorithm 3 Server response protocol in FLECKS: at server s_i, $1 \leq i \leq n$.

Variables:
List $L \in Tags \times \mathbb{N} \times \mathbb{F}_{256} \times \{Pre, Fin\}$,
initially empty.
Last Opnum received from each writer: $Op(w), w \in \mathcal{W}$
Final tuple $(t_f, opnum_f, c_{i,f})$, initially $(t_0, opnum_0, c_{i,0})$.
The set $\mathcal{R}$ of outstanding READ requests.
An element of $\mathcal{R}$ is the form $(r, t_{req}, opnum_{req})$. Initially, empty.

put-data-resp **received** $(opnum, c_i)$ **from** w :
  $Op(w) = \max(Op(w), opnum)$
/*change from Fin to Pre for writing of algorithm*/

  if $((w, \tilde{z}), opnum, \perp, Fin) \in L$ **then**
    $L \leftarrow L \cup \{((w, \tilde{z}), opnum, c_i, Pre)\}$
    Do *commit-tag*$((w, \tilde{z}), opnum)$
  else
    Let $t_{in} = (w, t_f.z + 1)$.
    $L \leftarrow L \cup \{(t_{in}, opnum, c_i, Pre)\}$.
    Send t_{in} to writer w.

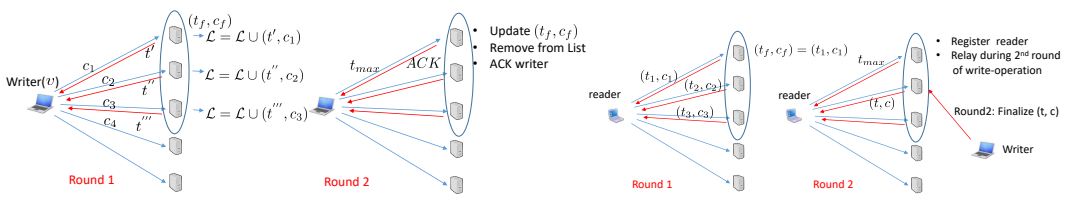
put-tag-resp **received** $(t, opnum)$ **from** w :
  Do *commit-tag*$(t, opnum)$
  Send ACK to writer w.

get-tag-data-resp **request received from** r :
  Send final tuple $(t_f, opnum_f, c_{i,f})$ to reader r

get-data-resp **received** $(t_{req}, opnum_{req})$ **from** r :
  $\mathcal{R} = \mathcal{R} \cup (r, t_{req}, opnum_{req})$.
  if $t_f \geq t_{req}$ **then**
    send $(t_f, opnum_f, c_{i,f})$ to reader r.
  Do *commit-tag*$(t_{req}, opnum_{req})$

read-complete-resp **request received from** r :
  $\mathcal{R} = \mathcal{R} \backslash (r, *, *)$.

commit-tag-resp$(t, opnum)$:
  Let $t = (w, z)$.
  if $((t.w, *), opnum, c_i, Pre) \in L$ **then**
    Update final tuple:
    if $t > t_f$ **then**
      $(t_f, opnum_f, c_{i,f}) \leftarrow (t, opnum, c_i)$.
    Relay: Send $(t, opnum, c_i)$ to every $r, (r, t_{req}, *) \in \mathcal{R}$ s.t. $t \geq t_{req}$.
    Remove from list: $L = L \backslash \{((w, *), opnum, c_i, *)\}$.
  else if $opnum > Op(t.w)$
    For Future: $L \leftarrow L \cup (t, opnum, \perp, Fin)$



(a) WRITE operation. **(b)** READ Operation.

Figure 2 High Level schematic overview of the WRITE and READ protocols of FELCKS.

The write Operation. Assume that a writer w wishes to WRITE (update to) value v. The writer computes the n coded elements $[c_1, \ldots, c_n]$. The WRITE operation consists of two rounds. At a high level, the first round is the temporary storage phase, where the server adds the coded element into the list. Once the writer gathers that k servers have done so, it starts the second round where a commit command is issued whereby the server updates the finalized tuple using the entry in the list (if the entry is newer). A pictorial overview of the WRITE protocol appears in Pseudocode 1. We now explain the two rounds in detail.

In the first round *put-data*, the writer sends the pair $(opnum, c_i)$ to server $s_i, 1 \leq i \leq n$, where *opnum* denote the writer's operation number for the ongoing WRITE operation. The server responds via *put-data-resp*. Upon receiving the message, under normal circumstances (the **else** part of the **if** statement), the server computes a new tag for this WRITE operation. This is obtained as $t_{in} = (w, t_f.z + 1)$, where t_f denotes the finalized tag stored by s_i, and $t_f.z$ denotes the integer part of the t_f. The server adds the tuple $(t_{in}, opnum, c_i, Pre)$ to the temporary storage list L, and responds to the writer by sending t_{in}. The **if** part of the pseudo-code is to take care of the rare case, when the message from the writer arrives too slow at the server, where the server has already learned by other means that the WRITE operation has already been *committed* by a quorum of servers. In this case, server s_i directly commits the message $(opnum, c_i)$ in round 1. The commit step, under normal circumstances, is part of the second round response of the WRITE operation, and is explained below. The writer waits to hear tags from k servers, and computes maximum z of the integer parts of the received tags. This completes round 1.

In the second round *put-tag*, the writer w creates the new tag $t = (w, z)$, and sends the pair $(t, opnum)$ to all servers. Upon receiving the message, a server performs, via *put-tag-resp*, the *commit-tag* step. Under normal circumstances (the **if** clause of *commit-tag-resp*), as part of the commit-tag-response, the server updates the finalized tuple with the entry in the list corresponding to $(t.w, opnum)$, if[3] $t > t_f$. The server also removes the entry from the list. This ensures that for any successful WRITE operation, every non-faulty server eventually automatically garbage-collects the temporary storage entry in the list. The **if** clause of the *commit-tag-resp* contains a *Relay* step that is used to server outstanding READ requests. This is explained as part of the READ operation below. The **else** part of *commit-tag-resp* step is executed during rare circumstances, when the server initiates the *commit-tag* step not as part of the round 2 of the corresponding WRITE operation, but learns from a reader that the WRITE operation has already begun the second round but this server has not even received the first round message form the writer yet. In the case, the server adds an indicator entry to the list L (using the forth Pre/Fin part of the entry), so that when the writer message arrives in future, the server can directly proceed to commit the coded-element. Finally, the writer terminates after receiving acknowledgments from k servers.

The read Operation. The reader during the first round contacts all the servers for the finalized tuples, and waits for responses from k servers. If all the responses have the same tag, clearly the reader can decode using the k responses, and the READ ends in the first round itself. Otherwise, the reader computes the maximum tag from among the tags received as part of the finalized tuples, and we call this the request tag t_{req}. The corresponding

[3] It is possible that the local temporary tag for corresponding the entry in list is higher than the received tag t. The reason is that the writer computes the tag by computing maximum among a quorum, and not all the servers. This local temporary tag is simply ignored, and the finalized tuple is saved using the tag received from the writer. The local temporary tag is used during the second round only to identify the correct entry in the list that must be committed.

$opnum_{req}$ is called request opnum. The goal in the second round is to use the relay-technique to let the reader decode a value corresponding to a tag that is at least as high as t_{req}. A pictorial overview of READ protocol appears in Pseudocode 2.

In the second round, the reader sends the pair $(t_{req}, opnum_{req})$ to all servers. Any server that receives the message *registers* the read-request, as part of the *get-data-resp* by adding the tuple $(r, t_{req}, opnum_{req})$ to the set $\mathcal{R}$ of outstanding READ requests. Further, if the finalized tag is at least as high as the request tag, the server sends finalized tuple to the reader. The goal of the reader registration is to enable *relaying* to the reader until the reader gathers k coded elements corresponding to some common tag. The relaying (to outstanding READ requests) happens whenever the server executes the *commit-tag-resp* step for a pair $(t, opnum)$ such that $t \geq t_{req}$. Recall that *commit-tag-resp* step is executed as part of the second round response of WRITE operations. It may be noted that a server only sends those (tag, coded-element) pairs that are committed, and thus form potential candidates for the finalized tuple. In this regard, from the point of view of the reader, the temporary storage list L can be thought as elongating the channel from the writer to the server such that a (tag, coded-element) pair is ready for consumption by the server only after the writer executes the second round.

As part of the *get-data-resp* step, the server also performs the *commit-tag* step for the pair $(t_{req}, opnum_{req})$. This is to handle the case where the writer crash fails half-way into the second round for the WRITE operation corresponding to $(t_{req}, opnum_{req})$. In this case, only a partial set of the servers would have performed *commit-tag* step for the pair $(t_{req}, opnum_{req})$, while the rest of the servers still hold the coded elements in the temporary storage list L. The execution of the *commit-tag* step as part of the READ operation is in spirit analogous to the reader-write-back (read-repair) operation performed replication algorithms [4], and helps complete a partially completed WRITE operation.

The reader collects (tag, coded-element) pairs until it receives k corresponding to a common tag, say t_r, whose corresponding value is decoded. During this process, if the reader receives a coded-element for a tag $t > t_{req}$, then (while waiting for further pairs), the reader sends out *commit-tag*$(t, opnum)$ message to the servers. The purpose of this commit tag is exactly the same as that of the *commit-tag*$(t_{req}, opnum_{req})$ described above. It may be noted that the utility of these messages only arise when the WRITE corresponding to $(t, opnum)$ failed half-way. Under normal circumstances, these messages are simply ignored by the server that has already seen the writer *commit-tag* message. In fact, as we shall see in the experiments, even with read-write ratio of 1, the number of reads needing the second round is a tiny fraction.

Finally, once the reader decodes, it sends a READ complete message so that the servers can stop relaying. Note that no responses are expected for these read-complete messages.

Handling Client Failures. While we show that FLECKS ensures linearizable executions and wait-freedom availability corresponding to non-faulty client processes despite failure of a reader or and writer process, we note that a failed reader/writer process introduces the need for additional intervention for performance optimization. A failed reader can result in servers relaying to the reader indefinitely. While it is definitely possible to stop relaying algorithmically as in [14] via a gossip protocol among the servers, the protocol is redundant for successful reads, and thus contributes high burden on the system from a practical point of view. Alternate practical solutions include letting the server stop the relaying after a certain timeout duration or threshold number relay messages. In fact, if point-to-point channel latency is bounded by Δ, any READ operation completes within 6Δ (see Section 4),

independent of the number of concurrent writes. In the rare event when the relaying stops even before the READ completes (when the point-to-point latency bound is not respected), one can always timeout the reader, and restart the read.

Similarly, a WRITE that fails during the first round leaves entries in the temporary storage list L that is not garbage collected by the algorithm. In our implementation, each server additionally garbage collects any entry in the list that is older than a certain threshold time that is set sufficiently high from a practical viewpoint.

4 Liveness and Atomicity of FLECKS

Liveness. Now we state and prove the liveness property of FLECKS. We recall that the algorithm uses an $[n, k]$ MDS code. We assume if a client has already started an operation (say π), the (same) client will wait until π is completed before starting a new operation.

▶ **Theorem 1.** *(**Liveness**) Consider any well-formed execution of FLECKS in which at most $f = n - k$ servers crash fail during the execution. Then, an operation corresponding to a non-faulty client completes irrespective of any past, ongoing or future successful or failed client operations.*

Proof. Liveness of a WRITE operation is easily verified from an inspection of the algorithm. For a READ operation, there is nothing to prove if the READ completes in the first round itself. The non-trivial part is proving liveness of a READ operation that executes the second phase. Let π be such a READ operation corresponding to reader r. As in the algorithm, let $(t_{req}, opnum_{req})$ denote the message sent by the reader during the *get-data* phase. Without loss of generality, let $s_1, \ldots, s_k$ denote the set of k servers that never fail during the execution. Let T_i denote the point of execution when s_i receives the *get-data* request from reader r. Let $T_{max} = max_{1 \leq i \leq k} T_i$. Next, let $t_i = s_i.t_f|_{T_{max}}$, i.e., t_i denotes the finalized tag stored by server s_i at T_{max}. Further, let $t_{max} = max_{1 \leq i \leq k} t_i$. The tags t_{max} and t_{req} are not necessarily ordered in any specific way. We now divide the discussion into the following cases:

Case a) $t_{max} \leq t_{req}$: In this case, we show that corresponding to every server $s_i, 1 \leq i \leq k$, there exists a point of execution $\hat{T}_i$ when s_i will send the message $(t_{req}, opnum_{req}, c_i)$ to reader r, unless s_i received read-complete message before $\hat{T}_i$. In this case, it is clear that the reader gets k coded elements corresponding to the tag t_{req} and thus, can definitely decode the value corresponding to t_{req}, after receiving the kth coded-element, unless the READ is complete even before. We consider two sub cases here:

Subcase i) Sever s_i did not receive put-data request with message $(t_{req}.w, opnum_{req}, c_i)$ until T_i: We know that the server s_i registers the READ request at T_i (by adding the corresponding entry to $\mathcal{R}$). Further, by assumption the channel from every writer to every server is ordered, and thus if the server has not received the *put-data* request with message $(t_{req}.w, opnum_{req}, c_i)$ until T_i, this means that $s_i.Op(w)|_{T_i} < opnum_{req}$. In this case, the server adds the tuple $(t_{req}, opnum_{req}, \perp, Fin)$ to its list as part of the execution of *commit-tag* step of *get-data-resp*. Let $\tilde{T}_i > T_i$ denote the point of execution when s_i receives *put-data* request with message $(t_{req}.w, opnum_{req}, c_i)$. Such a point in the execution necessarily exists because the tag t_{req} is committed tag, and thus at least one server received the *put-tag* request with message $(t_{req}, opnum_{req})$ directly from writer $t_{req}.w$. This means that the writer $t_{req}.w$ necessarily completed the *put-data* phase in which messages were sent to all n servers (since it already executed at least a part of the second phase). We recall here our channel model assumption that once message is placed in the channel, it is eventually delivered to the destination process, as long as the destination is non-faulty. In the current

proof, the server s_i is non-faulty, and thus will eventually receive $(t_{req}.w, opnum_{req}, c_i)$. This completes our justification of the existence of the point of execution $\tilde{T}_i$.

To continue with the proof, we note that during the *put-data-resp* action corresponding to $(t_{req}.w, opnum_{req}, c_i)$, server s_i finds that the WRITE operation has an entry in the list with Fin in the last field, and consequently executes *commit-tag* for the same WRITE operation. In this case, if s_i did not receive read-complete message until $\tilde{T}_i$, it is clear that server will relay the tuple $(t_{req}, opnum_{req}, c_i)$ to reader r, as part of the execution of *commit-tag-resp*$(t_{req}, opnum_{req})$. Note that in this case, we have $\hat{T}_i = \tilde{T}_i$.

Subcase ii) Sever s_i received put-data request with message $(t_{req}.w, opnum_{req}, c_i)$ before T_i: In this case, we first note that $s_i.t_f|_{T_i} \leq s_i.t_f|_{T_{max}} \leq t_{max} \leq t_{req}$. If $s_i.t_f|_{T_i} = t_{req}$, then the server sends the tuple $(t_{req}, opnum_{req}, c_i)$ to reader r as part of execution Step 2. of *get-data-resp* corresponding to message $(t_{req}, opnum_{req})$. If $s_i.t_f|_{T_i} < t_{req}$, then it is clear that s_i never received *commit-tag*$(t_{req}, opnum_{req})$ request until T_i, and hence it must be true that the tuple $(t_{req}.w, opnum_{req}, c_i) \in s_i.L|_{T_i}$. In this case, the tuple $(t_{req}.w, opnum_{req}, c_i)$ is relayed to the reader r as part of the execution of Step 3, *commit-tag-resp*$(t_{req}.w, opnum_{req})$, of the *get-data-resp* action.

Case b) $t_{max} > t_{req}$: In this case, we show that corresponding to every server $s_i, 1 \leq i \leq k$, there exists a point of execution $\hat{T}_i$ when s_i will send the message $(t_{max}, opnum_{max}, c_i)$ to reader r, unless s_i received read-complete message before $\hat{T}_i$. In this case, it is clear that the reader gets k coded elements corresponding to the tag t_{max} and thus, can definitely decode the value corresponding to t_{max}, after receiving the kth coded-element, unless the READ is complete even before.

To prove this, observe that there exists a server $s_j \in \{s_1, \ldots, s_k\}$ such that $s_j.t_f|_{T_{max}} = t_{max}$. We know that $T_j \leq T_{max}$, and hence $s_j.t_f|_{T_j} \leq s_j.t_f|_{T_{max}} = t_{max}$. If $s_j.t_f|_{T_j} = t_{max}$ (trivially true if $T_{max} = T_j$), the server s_j sends the tuple $(t_{max}, opnum_{max}, c_j)$ to reader r as part of the execution Step 2 of *get-data-resp*. If $s_j.t_f|_{T_j} < t_{max}$, it is clear that there exists a point of execution $\hat{T}_j, T_j < \hat{T}_j < T_{max}$, where server s_j executes *commit-tag-resp*$(t_{max}, opnum_{max})$ and changes the finalized tag to t_{max}. Thus, the server s_j relays the tuple $(t_{max}, opnum_{max}, c_i)$ to reader r at $\hat{T}_j$, if the server s_j has not yet received read-complete response. In summary, we have shown that there exists one server s_j among the set of non-faulty servers that will definitely send the tuple corresponding to $(t_{max}, opnum_{max})$ to the reader. Once the reader gets the first coded element corresponding to the pair $(t_{max}, opnum_{max})$, since $t_{max} > t_{req}$, the reader sends the *commit-tag*$(t_{max}, opnum_{max})$ message to all the servers.

It remains to be shown that every other server $s_i \in \{s_1, \ldots, s_k\} \backslash \{s_j\}$ also sends coded element corresponding to $(t_{max}, opnum_{max})$ to the reader. To show this, we once again observe that $s_i.t_f|_{T_i} \leq s_i.t_f|_{T_{max}} \leq t_{max}$. If $s_i.t_f|_{T_i} = t_{max}$, it is clear that the server s_i sends the tuple $(t_{max}, opnum_{max}, c_i)$ to reader r as part of the execution Step 2 of *get-data-resp*. Now consider the case $s_i.t_f|_{T_i} < t_{max}$. The READ request is clearly registered. From the discussion so far, we note that the server s_i will eventually receive both the *put-data* request corresponding to message $(t_{max}.w, opnum_{max}, c_i)$, and also the *commit-tag* request corresponding to message $(t_{max}, opnum_{max})$. The *put-data* request is eventually received since the writer has definitely completed the Phase 1 of the WRITE operation, and we know from the channel assumption that once a message is placed in the channel, it eventually arrives at the destination. The *commit-tag* request is eventually received since as observed above the reader sends the *commit-tag*$(t_{max}, opnum_{max})$ message to all the servers (useful if the writer failed during the execution of Phase 2 of the corresponding WRITE operation). Further, the algorithm is designed in such a way that the ordering of the arrivals of these two messages does not matter; arguments (using the Pre/Fin indicator) similar to those

used in Case a) can be used to show that the tuple $(t_{max}, opnum_{max}, c_i)$ is committed at the earliest point in the execution when both these messages are received. In this case, the server s_i relays the tuple corresponding to $(t_{max}, opnum_{max})$ to the reader, if s_i did not get *read-complete* message yet. This completes the proof of Case b), and hence the proof of liveness of a READ operation corresponding to a non-faulty reader. ◀

Atomicity. Below we state and prove the atomicity property of the FLECKS algorithm.

▶ **Theorem 2.** *(**Atomicity**) Any well-formed execution of FLECKS is atomic.*

Latency Analysis and Storage Cost. Although FLECKS is designed for asynchronous message passing settings, in the case of a reasonably well-behaved network we can bound the latency of an operation. Assume that any message sent on a point-to-point channel is delivered at the corresponding destination (if non-faulty) within a duration $\Delta > 0$, and local computations take negligible amount of time compared to Δ. Thus, latency in any operation is dominated by the time take taken for the delivery of all point-to-point messages involved. Under these assumptions, the latency bounds for successful WRITE and READ operations in FLECKS are as follows.

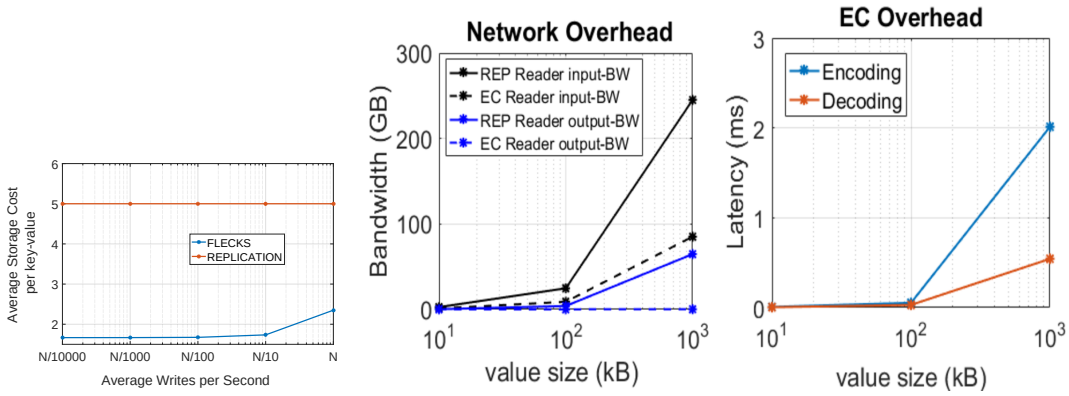
▶ **Theorem 3.** *The duration of a* WRITE *or a* READ *in FLECKS is at most* 4Δ *and* 6Δ*, respectively.*

Recall that READ operations use the technique of relaying for completion, and a new relay to the reader potentially occurs due to every concurrent WRITE operation. While this may happen, the above result guarantees a bound on the READ completion time that is independent of the number of concurrent writes experienced by the read.

Storage Costs. We now provide bounds on the total storage cost incurred by FLECKS under the bounded latency model. The storage cost at any point in the execution is the total amount of data that is stored in the servers. The cost at any server arises due to the storage of finalized coded-element as well as the storage of temporary coded-elements in the list - we account for both of these in our calculation. Costs contributed by meta-data are ignored while ascertaining either storage costs.

Consider a system storing N key-value pairs, where each pair is implemented via an instance of FLECKS. We assume using an $[n, k]$ MDS code for each of these instances. Further, every value is assumed to have the same size, and let us normalize it to 1 unit of space. Let ρ denote the average number of writes per second experienced by the system, where each WRITE can happen on any of the N objects allowing for concurrency. Further let θ denote the fraction of writes that fail (due to writer crashes). We know from the algorithm that the coded elements from such writes can potentially linger around in the temporary list until an external mechanism garbage collects them. Let τ denote the maximum duration for which any entry is retained in the list by a server - we assume that after τ seconds of adding an entry into the list, the server simply garbage collects the entry if it was not removed until then (automatically by the algorithm). The following theorem gives the average storage cost in the system in terms of the above parameters under the bounded latency model.

▶ **Theorem 4.** *The average storage cost per key-value pair incurred by a system running FLECKS under the bounded latency model is given by* $\frac{n}{k}\left[1 + \frac{(4\Delta+\theta\tau)\rho}{N}\right]$.



(a) Storage cost per object. **(b)** Bandwidth cost of reads **(c)** Delay due to encode or decode

Figure 3 (a) Average storage cost per object for ABD (5-way replication) and FLECKS using an $[n = 5, k = 3]$ erasure code is plotted as a function of number of WRITEs per second with $N = 10^4$ objects. Even for one WRITE per object per second, FLECKS significantly saves storage over ABD, for similar fault tolerance. (b) The total bandwidth consumed by each reader after executing $50,000$ reads. (c) The average latency to encode or decode a value. The plots are for runs with frequency of READ and WRITE is 1.

Proof. Cost at server s is given by $C_s = C_{s,1} + C_{s,2}$, where $C_{s,1}$ is the cost due to finalized entries, and $C_{s,2}$ is due to the entries in the list. The total storage cost C is then given by

$$C = \sum_s C_{s,1} + \sum_s C_{s,2} = N\frac{n}{k} + \sum_s C_{s,2}, \tag{1}$$

where Nn/k is the total cost in the system due to the finalized entries. Note that the total number of servers in the system does not appear anywhere in our analysis. To estimate the second term, we note that any point T in the execution, the average number of active writes retained by the system is given by $4\Delta\rho$. This follows because we know the from Theorem 3 that a WRITE completes within 4Δ seconds, and on average there are $4\Delta\rho$ writes that started within the time interval $[4\Delta - T, T]$ that remain active at time T. We also need to count the number of failed writes retained by the system at time T. The average number of failed writes retained by system at time T is given by $\tau\theta\rho$, and the argument is similar to the one for active writes. Thus, if $\sum_s C_{s,2}$ denotes the average cost due to the entries in the list across all servers, then this is given by $\sum_s C_{s,2} = \frac{(4\Delta\rho+\theta\tau\rho)n}{k}$. Now, the average cost per key-value pair in the system is given by $C/N = \frac{n}{k} + \frac{(4\Delta\rho+\theta\tau\rho)n}{Nk} = \frac{n}{k}\left[1 + \frac{(4\Delta+\theta\tau)\rho}{N}\right]$. ◀

An illustration of the storage cost bound is provided in Fig. 3 (a). In this example, we assume an $[n = 5, k = 3]$ code for a system storing $N = 10^4$ key-value pairs, where 0.01% of writes fail, i.e., $\theta = 10^{-4}$. We fix $\Delta = 100$ ms and $\tau = 100$ s, and these two numbers are based on observations from our own experiments. The storage cost is plotted as a function of writes per second in the system. For comparison, we also plot the storage cost that would be incurred by a 5-way replicated system.

5 Implementation and Experimental Validation

Here we briefly describe our experimental evaluation of FLECKS against an optimized version of the ABD algorithm. The algorithms (FLECKS and ABD) are implemented in Golang

version *go 1.6.3* with additional libraries for messaging (ZMQ [3]), erasure-coding (ISA-L [1]) and stats collection (libstatgrab [2]). The software is deployed via docker containers. For point to point communication among the processes, we use *ZMQ* 3.2.0 [3], which is a distributed (without a centralized broker) messaging library built on top of TCP/IP sockets. For the erasure-coding part of the implementation we use the open-source version of Intel's ISA-L [1]. We use the Cauchy matrix based MDS codes. We chose Galois field of size 256, since $GF(256)$ is fairly standard in the storage industry.

System Setting. We deployed each server and client process on a separate virtual machine (VM) running Ubuntu Linux 16.04 LTS configured with 8 GB of RAM and a 4-core CPU. The VMs were part on an OpenStack cloud platform. The bisectional bandwidth of the platform is about 10 Gbps.

In our experiments we stored up to 10000 atomic objects, where each object is implemented via an independent instance of FLECKS. Each server runs as a single threaded process handling all the objects associated with that server. A client process can access any of the objects. All data is stored in memory. For simulating crash failure of server process, we simply kill the process.

Latency of read and write operations. In Fig. 1, we plot average latency for reads and writes while accessing multiple objects (1, 10, 100, 1000 and 10000 objects) in executions of FLECKS and ABD. For this scenario, we use 5 readers, 5 writers, and 5 servers. We compare 5-way replication ABD with FLECKS based on $[5, 3]$ erasure-code. We notice that FLECKS has substantially reduction in latency and this improvement is more prominent as the size of payload increases.

Bandwidth cost for operations. Fig. 3(b) shows the total incoming and outgoing network bandwidth (BW) consumed by a single reader client in FLECKS and ABD. With 50000 operations and 5-way replication ABD, we expect incoming BW to be about 250 GB when object size is 1000 kB. From Fig. 1, we see that about 27% of that READs have two phases in ABD, and thus outgoing BW, dominated by two phase READs, is around $0.27 * \times 250 = 67$ GB. In FLECKS, the incoming BW is dominated by 1 phase READs, and is about $1/3 \times 250 = 83$ GB. Unlike replication, the 2 phase READs (roughly 3%) in FLECKS does not write-back actual data, and hence outgoing BW of FLECKS is negligible.

Latency due to encoding and decoding. Fig. 3(c) also shows the contribution of erasure code encoding and decoding time during a WRITE or a READ in FLECKS. Clearly, latency is minimally affected by the erasure-coding operations, consistent with other recent works in literature [20].

Server failures. To test the effect of server failures, we setup 1000 objects on 10 servers as in the experiment. After deployment, we kill two of the server processes (chosen at random). In agreement to our liveness guarantees the read and writes operations continue to complete. For a replicated system, increasing the number of replicas per object increases latency of operation.

Effect of Increasing Number of Readers. For a practical system, one expects to see a near-linear scaling of overall READ throughput against the number of readers. While we see this behavior for both replication and FLECKS, we noted that FLECKS permits a significantly better throughput scaling. The advantage can be directly attributed to the lower READ latency of FLECKS.

6 Conclusion

We investigated the feasibility of erasure-codes in atomic memory algorithms to reduce storage cost, bandwidth costs and latency. With that in mind We designed FLECKS for

asynchronous networks, that reduces, storage cost for the stored object and bandwidth cost for the operation. FLECKS completes the read operations in just one round in the absence of concurrent writes. FLECKS design is based on practical settings. FLECKS guarantees liveness of operations in the present of any client crash failures and up to $n-k$ server crashes. We proved the atomicity and liveness properties of FLECKS. We implemented FLECKS according to our algorithmic specifications. We performed extensive experiments on an actual network environment. Future work will invoke extending FLECKS to allow repair of crashed servers.

References

1. Intel® Intelligent Storage Acceleration Library (Intel® ISA-L). `https://software.intel.com/en-us/storage/ISA-L`. [Online; accessed 23-August-2018].
2. libstatgrab. `https://www.i-scream.org/libstatgrab/`. [Online; accessed 23-August-2018].
3. ZeroMQ: Distributed Messaging. `http://zeromq.org/`. [Online; accessed 23-August-2018].
4. H. Attiya, A. Bar-Noy, and D. Dolev. Sharing Memory Robustly in Message Passing Systems. *Journal of the ACM*, 42(1):124–142, 1996.
5. Christian Cachin and Stefano Tessaro. Optimal Resilience for Erasure-Coded Byzantine Distributed Storage. In *2006 International Conference on Dependable Systems and Networks (DSN 2006), 25-28 June 2006, Philadelphia, Pennsylvania, USA, Proceedings*, pages 115–124, Los Alamitos, CA, USA, 2006. IEEE Computer Society. `doi:10.1109/DSN.2006.56`.
6. Viveck R. Cadambe, Nancy A. Lynch, Muriel Médard, and Peter M. Musial. A coded shared atomic memory algorithm for message passing architectures. *Distributed Computing*, 30(1):49–73, 2017.
7. Yu Lin Chen Chen, Shuai Mu, and Jinyang Li. Giza: Erasure coding objects across global data centers. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC '17)*, pages 539–551, 2017.
8. Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM. `doi:10.1145/1294261.1294281`.
9. A. G. Dimakis, K. Ramchandran, Y. Wu, and C. Suh. A survey on network codes for distributed storage. *Proceedings of the IEEE*, 99(3):476–489, 2011.
10. Partha Dutta, Rachid Guerraoui, and Ron R. Levy. Optimistic Erasure-Coded Distributed Storage. In *DISC '08: Proceedings of the 22nd international symposium on Distributed Computing*, pages 182–196, Berlin, Heidelberg, 2008. Springer-Verlag. `doi:10.1007/978-3-540-87779-0_13`.
11. G.R. Goodson, J.J. Wylie, G.R. Ganger, and M.K. Reiter. In *Dependable Systems and Networks, 2004 International Conference on*. `doi:10.1109/DSN.2004.1311884`.
12. James Hendricks, Gregory R Ganger, and Michael K Reiter. Low-overhead byzantine fault-tolerant storage. *ACM SIGOPS Operating Systems Review*, 41(6):73–86, 2007.
13. W. C. Huffman and V. Pless. *Fundamentals of error-correcting codes*. Cambridge university press, 2003.
14. K. M. Konwar, N. Prakash, E. Kantor, N. Lynch, M. Médard, and A. A. Schwarzmann. Storage-Optimized Data-Atomic Algorithms for Handling Erasures and Errors in Distributed Storage Systems. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 720–729, May 2016.
15. Kishori M Konwar, N Prakash, Nancy Lynch, and Muriel Médard. RADON: Repairable atomic data object in networks. In *The International Conference on Distributed Systems (OPODIS)*, 2016.

16 Leslie Lamport. On Interprocess Communication, Part I: Basic Formalism. *Distributed Computing*, 1(2):77–85, 1986.

17 Leslie Lamport. Fast Paxos. *Distributed Computing*, 19:79–103, October 2006.

18 N Nicolaou, V Cadambe, N. Prakash, K.M. Konwar, M. Medard, and N Lynch. ARES: Adaptive, reconfigurable, erasure coded, atomic storage implementing a register in a dynamic distributed system. In *International Conf. on Distributed Computing Systems (ICDCS)*, 2019.

19 A. Spiegelman, Y. Cassuto, G. Chockler, and I. Keidar. Space Bounds for Reliable Storage: Fundamental Limits of Coding. In *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS2015)*, 2015.

20 Heng Zhang, Mingkai Dong, and Haibo Chen. Efficient and Available In-memory KV-Store with Hybrid Erasure Coding and Replication. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 167–180, 2016.

Interactive Coding Resilient to an Unknown Number of Erasures

Ran Gelles
Faculty of Engineering, Bar-Ilan University, Ramat-Gan, Israel
ran.gelles@biu.ac.il

Siddharth Iyer
University of Washington, WA, USA[1]
siyer@cs.washington.edu

Abstract

We consider distributed computations between two parties carried out over a noisy channel that may erase messages. Following a noise model proposed by Dani et al. (2018), the noise level observed by the parties during the computation in our setting is arbitrary and a priori unknown to the parties.

We develop *interactive coding schemes* that adapt to the actual level of noise and correctly execute any two-party computation. Namely, in case the channel erases T transmissions, the coding scheme will take $N + 2T$ transmissions using an alphabet of size 4 (alternatively, using $2N + 4T$ transmissions over a binary channel) to correctly simulate any binary protocol that takes N transmissions assuming a noiseless channel. We can further reduce the communication to $N + T$ by relaxing the communication model and allowing parties to remain silent rather than forcing them to communicate in every round of the coding scheme.

Our coding schemes are efficient, deterministic, have linear overhead both in their communication and round complexity, and succeed (with probability 1) regardless of the number of erasures T.

2012 ACM Subject Classification Theory of computation → Interactive computation; Mathematics of computing → Coding theory; Computing methodologies → Distributed algorithms

Keywords and phrases Interactive coding, erasure channels, distributed computation with noise, unbounded noise

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.13

Related Version A preliminary full version of this work [18] is available at `https://arxiv.org/abs/1811.02527`.

Funding *Ran Gelles*: Research supported in part by the Israel Science Foundation (ISF) through grant No. 1078/17.

Acknowledgements We thank Amir Leshem for plenty of helpful discussions.

1 Introduction

Consider two remote parties that use a communication channel in order to perform some distributed computation. One main obstacle they may face is noise added by the communication channel, corrupting their messages and ruining the computation. In the early 90's, Schulman [29, 30] initiated the field of *interactive coding* where the parties use coding techniques in order to complete their computation correctly despite possible communication noise.

Channels may introduce different types of noise. Among the common noise types are *substitution* noise, where the channel changes the content of messages (e.g., it flips communicated bits), *insertions and deletions*, where the channel may introduce a new message

[1] Part of this work was done while at Bar-Ilan University and a student at Birla Institute of Technology and Science, Pilani, India.

© Ran Gelles and Siddharth Iyer;
licensed under Creative Commons License CC-BY
23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller; Article No. 13; pp. 13:1–13:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

or completely remove a transmission, and *erasures* where the channel erases transmissions, i.e., replacing them with an erasure mark. Throughout the last several years many interactive coding schemes were developed, allowing parties to perform computations over the various channels and noise types, e.g., [9, 12, 32], see related work below and [17] for a survey.

Naturally, some bounds on the noise must be given. As a trivial example, it is clear that if *all* the transmissions are corrupted, there is no hope to complete any distributed computation. Braverman and Rao [9] showed that for substitution noise, a noise fraction of $1/4$ is maximal. That is, as long as the noise is limited to corrupt less than $1/4$ of the communication, there are coding schemes that will succeed in producing the correct output. However, if the noise exceeds this level, *any* coding scheme for a general computation is bound to fail. Similarly, noise rate of $1/4$ is maximal (and achievable) for insertion and deletion noise [7, 32], and noise rate of $1/2$ is maximal (and achievable) for erasure noise [14, 12].

All the above schemes must know *in advance* the amount of noise they are required to withstand. For instance, the scheme of [9] is given some parameter $\rho < 1/4$, which stands for the (maximal) fraction of noise that instance will be able to handle. Given this parameter and the length of the (noiseless) computation to be performed N, the coding scheme determines how many transmissions it should take in order to perform the computation, say, $\tilde{N} = \tilde{N}(\rho, N)$ transmissions. It is then guaranteed that as long as the noise corrupts at most $\rho\tilde{N}$ transmissions, the computation succeeds.

In a recent work, Dani et al. [11] considered the case where the noise amount may be arbitrary and *a priori* unknown to the coding scheme. That is, the channel may corrupt up to T transmissions, where $T \in \mathbb{N}$ is some fixed amount of noise which is independent of the other parameters of the scheme and the computation to be performed. The work of Dani et al. [11] considered substitution noise and showed a scheme that succeeds with high probability and, if the channel corrupts T transmissions during the execution of the scheme, then the scheme will take $N + O(T + \sqrt{TN \log N})$ transmissions to conclude.

Again, some limitations must be placed on the noise. Indeed, assume that an uncorrupted computation on the input (x, y) terminates in $\tilde{N}$ rounds, and assume $T > \tilde{N}$. Then, a substitution noise can always make Alice (wlog) believe that Bob holds y, by corrupting the messages Alice receives in the first $\tilde{N}$ rounds. Note that Alice then terminates with the wrong output, i.e., the output for (x, y). Dani et al. [11] dealt with the above impossibility by assuming that the parties have access to some shared randomness and that the adversary is *oblivious* to that randomness. This allowed them to employ cryptographic tools in order to guarantee the authentication of communicated messages.

In this work we focus on *erasure noise*, where the channel either transmits the message as is, or erases the message and outputs a special $\perp$ symbol that indicates this event. Erasure noise naturally appears in many practical situations, e.g., when an Ethernet packet gets corrupted yet this corruption is detected by the CRC checksum mechanism [25]. In this case the packet is considered invalid and will be dropped. This situation is equivalent to an erasure of that transmission. We note that this type of noise is weaker than substitution noise. This allows us to obtain coding schemes without further assumptions, mainly, without the need for shared randomness and without requiring the adversary to be oblivious.

Our main result is an efficient and deterministic coding scheme that withstands an arbitrary and *a priori* unknown T erasures with probability 1.

▶ **Theorem 1** (main, informal). *Given any two-party binary interactive protocol π of length N, there exists an efficient and deterministic noise-resilient protocol Π_4 of length $N + 2T$ that simulates π over a 4-ary channel in the presence of an arbitrary and a priori unknown number T of erasures.*

Alternatively, there exists an efficient, deterministic, binary *noise-resilient protocol* Π_2 *of length* $2N + 4T$ *that simulates* π *assuming an arbitrary and a priori unknown number* T *of erasures. It holds that*

$$\mathsf{CC}(\Pi_4) = \mathsf{CC}(\Pi_2) = 2N + 4T.$$

Since T, the amount of noise, is unknown to begin with, the length of the coding scheme must adapt to the actual noise that the parties observe. Such coding schemes are called *adaptive* [3]. Adaptivity raises several issues that must be dealt with appropriately. The main issue is *termination*. Since the coding scheme adapts its length to the observed noise, and since the different parties observe different noise patterns, their termination is not necessarily synchronized. As a matter of fact, obtaining synchronized termination is impossible.

▶ **Lemma 2.** *Let* Π *be a protocol for exchanging messages between Alice and Bob, that is resilient to an unbounded amount* T *of erasures. Then, there always exists a noise pattern for which Alice and Bob terminate in different rounds.*

This can be seen as a variant of the famous "coordinated attack problem" [24], where reaching full synchronization between two parties is known to be impossible. See [18] for a proof and an elaborated discussion about unsynchronized termination.

Unsynchronized termination means that one party may terminate while the other party continues to send (and receive) transmissions as dictated by its protocol. In this case, the communication model should specify what happens in those rounds where only one party is active and the other has terminated. Specifically, it should specify what messages the active party receives in this case.

In our setting we define a special symbol we call *silence* (cf. [3]). We assume that silence is (implicitly) communicated by a terminated party. That is, the still-active party hears silence in every round it is set to listen and the other party has already terminated. We note that silence is corruptible – the channel may erase silent transmissions, and the active party will see an erasure mark instead. On the other hand, these implicit silent transmissions are not considered part of the communication of the protocol (i.e., we do not count them towards the communication complexity). Hearing a silence is a univocal indication that the other side has terminated, allowing the other party to terminate as well and bypassing the impossibility of synchronized termination.

In the setting of Theorem 1 we do not allow the parties to remain silent before termination – in each timestep a party is set to speak it must communicate a valid message. However, in today's networks, especially in networks with multiple parties, it is very common that parties send messages only if they have information to send, and keep silent otherwise.

Our second result extends Theorem 1 to the setting where parties are allowed to either speak or remain silent at every timestep (called the AGS setting hereinafter, see [3]). In this case, termination becomes even more tricky. Recall that the coding scheme of Theorem 1 uses silence as an indicator for termination. We can do the same in the AGS setting and avoid using silence throughout the protocol, keeping it as an indicator towards termination only. However, this would effectively reduce the AGS setting to the one of Theorem 1, and lead to a suboptimal scheme.

Instead, we take a different approach that requires parties to remain silent during the protocol in certain cases. This has the effect of reducing the communication at the expense of not being able to identify termination at times. In particular, one of the parties may remain active indefinitely. However, that party will remain silent after the other party has terminated, and moreover, it will hold the correct output of the computation. We call this situation *semi-termination*.

▶ **Definition 3.** *We say that a protocol has semi-termination if there exists a round t after which both the following conditions hold:*

1. *Both parties have computed the correct output, and*
2. *Both parties remain silent indefinitely (whether they terminate or not)*

Our result for this setting is as follows.

▶ **Theorem 4** (AGS setting, informal)**.** *Given any binary two-party interactive protocol π of length N, there exists an efficient and deterministic protocol Π_4 in the AGS setting with semi-termination, of length $N + 4T$ sending at most $N + T$ symbols from alphabet of size 4, that simulates π assuming an arbitrary and a priori unknown number T of erasures.*

Alternatively, there exists an efficient and deterministic protocol Π_1 in the AGS setting with semi-termination, with length $4N + 16T$ that communicates at most $N + T$ unary (i.e., non-silent) symbols and simulates π assuming an arbitrary and a priori unknown number T of erasures.

In this setting, it is significant to bound the round complexity since silence is not counted towards the communication complexity, yet it can be used to transfer information. As stated in the above theorem, the round complexity of the resilient protocols Π_4 and Π_1 until the event of semi-termination, is linear in the round complexity of the noiseless protocol π, namely, $\mathsf{RC}(\Pi_4) = O(N + T)$ and $\mathsf{RC}(\Pi_1) = O(N + T)$.

Organization. In the next part we overview the techniques used to obtain Theorem 1 and Theorem 4. Section 2 formally defines the communication and noise model, and fixes some notations used throughout. In Section 3 we describe the noise resilient protocol of Theorem 1 and analyze its correctness. The coding scheme of Theorem 4 along with its correctness is deferred to the full version of this paper (see [18] for a preliminary version). Further, in [18] we discuss the issue of unsynchronized termination and prove that it is impossible to obtain synchronized termination when the noise is unbounded.

1.1 Coding Schemes Overview

Erasure noise has two attractive properties we utilize towards our scheme. The first is that, if there was a corruption, the *receiver* is aware of this event; the second property is that, if there was no corruption, the received message is the *correct* one. Our scheme follows a technique by Efremenko et al. [12], where the parties simulate the noiseless protocol π bit by bit. As long as there is no noise, they can carry out the computation identically to π. However, in the case of an erasure, the receiver needs to signal the other side that it did not receive the last message and request a retransmission. The main problem is that this request message may get erased as well, making both sides confused regarding to what should be sent next.

In [12] this issue is solved by extending each message by two bits that indicate the round currently being simulated. It is proven that the parties may simulate different rounds, however, the discrepancy in the round number is bounded by 1. Hence, a round number modulus 3 is required in [12] to indicate whether Alice is ahead, Bob is ahead, or they are at the same round.

Our scheme combines the above technique with a challenge-response technique employed by Dani et al. in [11], in order to obtain resilience against an unbounded number of erasures. Our coding scheme is *not* symmetric, but rather Alice always begins a round by sending a "challenge" message, followed by Bob replying with a response. Alice then determines

whether the challenge-response round was successful: if both messages were not erased, Alice would see the correct response from Bob and would deduce that both messages were received correctly. If Alice received an erasure, or if she received the wrong response, she would deduce that an erasure has occurred during this round and the round should be re-simulated.

Bob, similarly but not identically, gets the challenge message from Alice and verifies that it belongs to a new round (that is, the challenge differs from the previous round). If this is the case, he replies with the next bit. Otherwise, i.e., if Alice's challenge was erased, or if Bob receives the challenge of the previous round, he replies with the response of the *previous* round. Note that if Alice did send a new challenge and it was erased, she will now get the response of the previous round and realize there is a mismatch.

It is not too difficult to see that the "challenge" suffices to be a single bit – the current simulated round number in π, modulus 2 (which we call the *parity* from now on). Since the scheme is not symmetric, it can never happen that Alice has advanced to the next round while Bob has not. The other case, where Bob is ahead of Alice by a single round is still possible. Therefore, one bit of information suffices to distinguish between these two cases.

In more details, Alice begins a round by sending Bob the next bit of the simulation of π, along with the parity of the round number (in π) of that bit. If Bob receives this message and the parity matches the round number he expects, he records the bit sent by Alice and replies with the next bit of π using the same parity. If this message reaches Alice correctly, she knows the round is over and advances to the next round. If Bob does not receive Alice's message (i.e., it gets erased), or if the parity is incorrect, Bob replies with the bit of the previous round along with the parity of that round. Similarly, if Alice receives a message with a wrong parity or an erased message, she keeps re-simulating the same round, until she gets the proper reply from Bob.

Note that a single erasure delays the progress of the simulation by a single round (2 transmissions). However, once there is a round in which both messages are not erased, the simulation correctly continues, and the succeeding two bits of π are correctly simulated. That is, as long as there is noise, the simulation just hangs, and when the noise ceases, the simulation continues from exactly the same place it stopped.

Once Alice completed simulating the last round of π, she quits. Recall that Bob is always ahead of Alice, thus if Alice completes the simulation, so does Bob. After Alice terminates, Bob receives silence unless erased by the channel. When Bob hears a silence, he learns that Alice has terminated and quits the scheme as well. The noise may delay Bob's time of termination by corrupting the silence, however, once the noise is over, Bob will learn that the simulation has completed and will quit as well.

Coding schemes for parties who may keep silent

Our second scheme works in the communication model where parties are allowed to remain silent if they wish (the AGS setting). The main advantage in being able to remain silent is allowing the parties to communicate information in an optimized manner which reduces their communication complexity. Specifically, consider the above idea of challenge-response, where a party replies with the wrong response in order to indicate there was an error and that a round should be re-simulated. In the AGS setting we can instead keep silent in order to signal this retransmission request.

The idea is as follows. Similar to the scheme above Alice begins a round by sending her bit to Bob (along with the parity). If the transmission is received correctly, Bob replies with his next bit. If Bob receives an erasure instead, he remains silent. This signals Alice that an

error has happened and that she should re-simulate the last round. Similarly, if Alice sees an erasure she keeps silent. In all other cases, i.e., receiving a silence or receiving a wrong parity, the parties re-transmit their last message as before.

The effect of a party keeping silent for asking a retransmission is reducing the communication complexity. Note that each erasure causes the recipient to remain silent for one single round, instead of sending a message that indicates an erasure. Then, the simulation continues from the point it stopped. On the other hand, the analysis becomes slightly more difficult in this case since silence may be erased as well, causing the other side to remain silent and signal there was an erasure. This may cause the first party to repeat its message while the other side should have actually resent its message in order to advance the simulation. Luckily, this issue does not falsify the correctness of the simulation – as a result of sending the parity, the extra retransmission is simply ignored. Furthermore, such a superfluous transmission does not increase the communication overhead since it can only happen when multiple erasures have occurred in the same round or in consecutive rounds.

When silence has a meaning of requesting a retransmission, we cannot use it anymore to indicate termination. Note that whatever Alice sends Bob to inform him she is going to terminate may get erased, so if Alice terminates Bob will not be aware of this fact and will remain active. If Alice waits to hear a confirmation from Bob that he received the indication and learned that Alice is about to terminate – this confirmation may get erased. Bob never learns if Alice has received his acknowledgment or not; then, if Bob assumes that Alice has terminated and terminates himself, it will be Alice who hangs in the protocol waiting for Bob's confirmation, and so on. Our approach to this conundrum is to allow the parties not to terminate as long as there exists a point in time beyond which the parties remain silent and both hold the correct output. In our scheme, Alice will actually terminate once she learns that the simulation is done[2] (recall that if Alice completed the simulation then Bob completed it as well, but the other direction does not necessarily hold). Bob's actions in the final part of the protocol are slightly different from his normal behaviour. Once Bob completed simulating π, but he is unaware whether or not Alice completed simulating π, he keeps silent unless he hears a message from Alice that re-simulates the final round. In this case, he replies with his final bit. In all other cases he remains silent.

1.2 Related Work

The field of interactive coding was initiated by the seminal work of Schulman [29, 30] focusing on two parties that communicate over a binary channel with either substitution noise (random or adversarial) or with erasure noise. Followup work (for substitution noise) developed coding schemes with optimal resilience [9, 6, 20] efficiency [31, 8, 19, 4, 20, 16], or good rate [26, 22, 16]. Coding schemes for different channels and noise types were developed in [28, 12, 15] for channels with feedback, in [7, 32, 23, 13] for insertions and deletions noise, and in [5, 27] for quantum channels. Interactive coding over channels that introduce erasure noise was explored in [30, 28, 14, 15, 12, 3]. In particular, Efremenko et al. [12] developed efficient coding schemes for optimal erasure rates, and Gelles and Haeupler [15] developed efficient coding schemes with optimal rate assuming a small fraction of erasures. Adaptive models were considered in [3, 21, 20]. See [17] for a survey on the field of interactive coding.

[2] It is possible to let Alice send Bob a "termination message" right before she quits in order to signal Bob she is about to terminate. In case this message is not erased, Bob will terminate as soon as he hears this special message. Otherwise, he will keep running the scheme but remain silent. In either case, the protocol satisfies the semi-termination requirements.

Closest to our work is the work of Dani et al. [11] who considered the case of an arbitrary noise amount that is unknown to the scheme. Their coding scheme assumes substitution noise which is oblivious to the randomness used by the parties as well as to the bits communicated through the channel. An AMD code [10] is used to fingerprint each transmitted message, allowing the other side to detect corruptions with high probability. Aggrawal et al. [2] use similar techniques to develop a robust protocol for message transfer, assuming bi-directional channel that suffers from an arbitrary (yet finite) and unknown amount of bit flips. Aggarwal et al. [1] extended this setting to the multiparty case, where n parties, rather than two, perform a computation over a noisy network with an arbitrary and unknown amount of noise.

2 Model Definition

Standard notations. For an integer $i \in \mathbb{N}$ we denote by $[i]$ the set $\{1, 2, \ldots, i\}$. All logarithms are taken to base 2. The concatenation of two strings x and y is denoted $x \circ y$. We let λ denote the empty string.

Interactive computation. In our setting, two parties, Alice and Bob, possess private inputs $x \in \{0,1\}^n$ and $y \in \{0,1\}^n$, respectively, and wish to compute some predefined function $f(x, y)$. The computation is performed by exchanging messages over a channel with fixed alphabet Σ. The computation is specified by a synchronous *interactive protocol* π. An interactive protocol $\pi = (\pi_A, \pi_B)$ is a pair of algorithms that share a common clock, and specify, for each timestep and each one of the parties, the following details: (1) which party speaks and which party listens in this time-step; (2) if the party is set to speak, which symbol to communicate; (3) whether the party terminates in this timestep, and if so, what the output is.

Without loss of generality, we assume that the (noiseless) protocol π is alternating, i.e., Alice and Bob speak in an alternating manner, Alice speaks in odd rounds and Bob in even rounds. If this is not the case, it can be made so while increasing the communication by a factor of at most 2. We define a *round* of the noiseless protocol to be a sequence of two time steps in which two consecutive messages are sent: the first is sent by Alice and the second by Bob. For example, the first round consists of the first message sent by Alice and the subsequent message from Bob. In general for $r \geq 1$, after $r - 1$ rounds have elapsed, the rth round consists of the rth message sent by Alice and Bob's subsequent message. For the sake of convenience, we assume that the last message of the protocol is sent by Bob; this can be ensured by padding the protocol by at most one bit. Since the protocol is alternating we can also think of rounds being associated with timesteps. More formally, in the rth round, Alice and Bob send messages corresponding to the $(2r-1)^{th}$ and the $2r^{th}$ timestep respectively. We say that a protocol has length N if Alice and Bob exchange N messages in the protocol.

Given a specific input (x, y) the transcript $\pi(x, y)$ is the concatenation of all the messages received during the execution of π on (x, y).

Erasures. The communication channel connecting Alice and Bob is subject to *erasure* noise. In each timestep, the channel accepts a symbol $s \in \Sigma$ and outputs either s or a special erasure mark ($\perp \notin \Sigma$). The noise is assumed to be worst-case (adversarial), where up to T symbols may be replaced with erasure marks. The value of T is unbounded and *unknown* to the parties – their protocol should be resilient to any possible $T \in \mathbb{N}$. A *noise pattern* is a bit-string $E \in \{0,1\}^*$ that indicates erasures in a given execution instance of the protocol. Specifically, there is an erasure in the i-th timestep if and only if $E_i = 1$. Given a specific

instance where both parties terminate before its s-th transmission, the number of erasures that are induced by E on that instance is the Hamming weight of $E_1, \ldots, E_s$, i.e., the number of 1's in the bit-string. We sometime allow E to be of infinite length, however our protocols will be resilient only against noise patterns with bounded amount of noise (i.e., if E is infinite, then its suffix is required to be the all-zero string; we call such noise *finite* noise).

Coding scheme: Order of speaking, silence and termination. A *coding scheme* is a protocol Π that takes as an input another protocol π (that assumes noiseless channels) and simulates π over a noisy channel. By saying that Π *simulates* π over an erasure channel with (unbounded) T corruptions, we mean the following. For any pair of inputs (x, y) given to Alice and Bob, after executing Π in the presence of T erasures, Alice and Bob will produce the transcript $\pi(x, y)$. The above should hold for *any* $T \in \mathbb{N}$, independent of π, x, y and unknown to Π (i.e., to the parties).

We assume that the protocol Π, at any given timestep, exactly one party can be the sender and the other party is the receiver. That is, it is never the case that both parties send a symbol or both listen during the same timestep. On the other hand we do not assume that the parties terminate together, and it is possible that one party terminates while the other does not. In this case, whenever the party that hasn't yet terminated is set to listen, it hears some default symbol $\square$, which we call *silence*.

There are several ways to treat silence. One option, taken in [3], is to treat the silence similar to any other symbol of Σ. That is, as long a party has not terminated and is set to speak, it can either send a symbol or remain silent ("send $\square$") at that round.[3] A different approach would be to require the parties to speak a valid symbol from Σ while they haven't terminated. That is, a party cannot remain silent if it is set to speak; this prevents the parties from using silence as a means of communicating information during the protocol. Once a party terminates, and only then, silence is being heard by the other party. This mechanism makes it easier for the parties to coordinate their termination. In particular, the event of termination of one party transfers (limited) information to the other party. We take this approach in the scheme of Section 3.

Another subtlety stems from the fact that the length of the protocol is not predetermined. That is, the length of the protocol depends on the actual noise in the specific instance. Such protocols are called adaptive [3]. In this case it makes sense to measure properties of the protocol with respect to a specific instance. For instance, given a specific instance of the protocol Π on inputs (x, y) with some given noise pattern, the communication complexity $\mathsf{CC}(\Pi(x, y))$ is the number of symbols *sent* by both parties in the specific instance. The communication is usually measured in bits by multiplying the number of symbols by $\log |\Sigma|$. The noise in a given instance is defined to be the number of corrupted transmissions until both parties have terminated, *including* corruptions that occur after one party has terminated and the other party has not. Corruptions made after both parties have terminated cannot affect the protocol, and we can assume such corruptions never happen.

3 A coding scheme for an unbounded number of erasures

In this section we provide a coding scheme that takes as an input any noiseless protocol π and simulates it over a channel that suffers from an unbounded and unknown number of erasures T. The coding scheme uses an alphabet Σ of size 4; in this setting parties are not allowed to be silent (unless they quit the protocol) and in every round they must send a symbol from Σ.

[3] We take this approach in the coding scheme designed to prove Theorem 4, see [18].

3.1 The Coding Scheme

The coding scheme for Alice and Bob, respectively, is depicted in Algorithms 1 and 2.

Inspired by the simulation technique of [12], our simulation basically follows the behavior of the noiseless protocol π step by step, where Alice and Bob speak in alternating timesteps. In each timestep, the sending party tries to extend the simulated transcript by a single bit. To this end, the parties maintain partial transcripts $\mathcal{T}^A$ for Alice (and $\mathcal{T}^B$ for Bob) which is the concatenation of the information bits of π that the parties have simulated so far and are certain of. Then, if Alice is to send a message to Bob, she generates the next bit of π given her partial simulated transcript, i.e., $\pi(x \mid \mathcal{T}^A)$, and sends this information to Bob.

In addition to the information bit, Alice also sends a parity of the round number she is currently simulating. That is, Alice holds a variable r_A which indicates the round number she is simulating. Recall that each round contains two timesteps, where Alice communicates in the first timestep and Bob in the second. Alice sends Bob the next bit according to her $\mathcal{T}^A$ and waits for Bob's reply to see if this round was successfully simulated. If Bob's reply indicates the same parity (i.e., the same round), Alice knows her message arrived to Bob correctly and hence the round was correctly simulated. In this case Alice increases r_A. Otherwise, she assumes there was a corruption and she keeps r_A as is; this causes the same round of π to be re-simulated in the next round of the simulation protocol.

Bob holds a variable r_B which again holds the (parity of) the latest round in π he has simulated. In a somewhat symmetric manner (but not identical to Alice!), he expects to receive from Alice the bit of the *next* round of π, $r_B + 1$. If this is the case he responds with his bit of that same round, or otherwise he re-transmits his bit of round r_B.

Our coding scheme assumes a channel with alphabet $\Sigma = \{0,1\} \times \{0,1\}$, where every non-silent message can be interpreted as $m = (\mathsf{Info}, \mathsf{Parity})$, where $\mathsf{Info} \in \{0,1\}$ is the information bit (simulating π) and $\mathsf{Parity} \in \{0,1\}$ is the parity of the round of π simulated by the sender.

The above continues until Alice has simulated the entire transcript of π, i.e, when r_A reaches the number of rounds in π. At this point, Alice exits the protocol. Bob, however, cannot tell whether Alice has completed the simulation or not and waits until he sees a silence, which indicates that Alice has terminated, only then does he exit the protocol. As regards the correctness, we prove that in any round of the simulation, Bob has seen at least as much of the noiseless protocol that Alice herself has seen. In particular, when Alice exits, Bob must have seen the entire transcript of the noiseless protocol.

3.2 Analysis

Preliminaries. Recall that in our terminology a round consists of two timesteps, where at every timestep one party sends one symbol from Σ. Alice sends symbols in odd timesteps and Bob in even ones. The above applies for both the noiseless protocol π (where the alphabet is binary) and for the coding scheme Π given by Algorithms 1 and 2 (where $\Sigma = \{0,1\} \times \{0,1\}$).

We think of the communication transcript as a string obtained by the concatenation of symbols sent during the course of the protocol run. Given a noiseless protocol π and inputs x, y, we denote by $m_A^\pi(i)$ (respectively, $m_B^\pi(i)$) the message sent by Alice (respectively, Bob) in the i-th round in the noiseless protocol π. Let $\mathcal{T}^\pi(r)$ be the transcript of the players after r rounds in π. The length of π is denoted N and without loss of generality we will assume that N is even.

We start our analysis by fixing a run of the coding scheme Π, specified by fixing an erasure pattern and inputs (x, y). Let k be the number of timesteps in the given run, and note that k is always odd. We define rounds in the coding scheme Π in the same way as we

Algorithm 1 Simulation over Erasure Channel with Unbounded Noise (Alice).

Data: An alternating binary protocol π of length N and an input x.

A.1 *Initialize* $\mathcal{T}^A \leftarrow \lambda$, $r_A \leftarrow 0$.
A.2 **while** $(r_A < \frac{N}{2})$ **do**
    `// Send a Message (odd time-step)`
A.3 $r_A \leftarrow r_A + 1$
A.4 $b_{send} \leftarrow \pi(x \mid \mathcal{T}^A)$
A.5 $\mathcal{T}^A \leftarrow \mathcal{T}^A \circ b_{send}$
A.6 send $(b_{send}, r_A \mod 2)$

    `// Receive a Message (even time-step)`
A.7 Obtain $m' = (b_{rec}, r_{rec})$
A.8 **if** $m' \neq \perp$ ***and*** $r_{rec} = r_A \mod 2$ **then**
A.9 $\mathcal{T}^A \leftarrow \mathcal{T}^A \circ b_{rec}$
    else
A.11 Delete last symbol of $\mathcal{T}^A$
A.12 $r_A \leftarrow r_A - 1$
    end
end
A.13 Output $\mathcal{T}^A$

define rounds in the noiseless protocol. That is, for $i \in \{1, 2, \ldots, \lceil \frac{k}{2} \rceil\}$, round i (in the coding scheme) corresponds to the timesteps $2i - 1$ and $2i$. Throughout this section by round i we refer to the round in the coding scheme unless it is specified otherwise. For the sake of clarity, we maintain that round i begins at the start of the $(2i - 1)$-th timestep and ends at the ending of the $2i$-th timestep. Since Π is alternating, Alice begins a round by sending a message and the round ends after Alice receives Bob's response.

We define t_A (and t_B) to be the termination round of Alice (and respectively, Bob). Observe that $t_A < t_B = \lceil \frac{k}{2} \rceil$ since Bob terminates only once he hears silence, which can happen only if Alice has already terminated in a previous round. For any round i and variable v we denote by $v(i)$ the value of v at the beginning of round i. In particular, $r_A(i)$ and $r_B(i)$ are the values of r_A and r_B at the beginning of round i. Since r_A is not defined after Alice exits in Algorithm 1, for all $t_A < i \leq t_B$, we set $r_A(i)$ to be the value of r_A at the *end* of round t_A, that is, its value just before Alice quits. We similarly define $\mathcal{T}^A(i)$ for $t_A < i \leq t_B$. Finally, let $m_A(i) = (b_A(i), \rho_A(i))$ and $m_B(i) = (b_B(i), \rho_B(i))$ be the messages sent by Alice and Bob (see Lines A.6 or B.13 or B.15) respectively in round i.

Technical lemmas and proof of correctness. The main technical claim of this part is Lemma 7, which we will now prove. We begin with the simple observation that, in every round, the parties' transcripts (and the respective round number the parties believe they simulate) either increase by exactly the messages exchanged during the last round, or they remain unchanged.

▶ **Lemma 5.** *For any $i \in [t_B]$, the following holds.*

1. $r_A(i+1) \in \{r_A(i), r_A(i) + 1\}$ *and* $\mathcal{T}^A(i+1) \in \left\{\mathcal{T}^A(i), \mathcal{T}^A(i) \circ b_A(i) \circ b_B(i)\right\}$ *and,*
2. $r_B(i+1) \in \{r_B(i), r_B(i) + 1\}$ *and* $\mathcal{T}^B(i+1) \in \left\{\mathcal{T}^B(i), \mathcal{T}^B(i) \circ b_A(i) \circ b_B(i)\right\}$.

Furthermore, r_A (r_B) changes if and only if $\mathcal{T}^A$ ($\mathcal{T}^B$) changes.

Algorithm 2 Simulation over Erasure Channel with Unbounded Noise (Bob).

```
      Data: An alternating binary protocol π of length N and an input y.
B.1   Initialize T^B ← λ, r_B ← 0, err ← 0, m ← (0,0)
B.2   while m' ≠ □ do                                  // while Silence isn't heard
        // Receive a Message (odd time-step)
B.3     Obtain m' = (b_rec, r_rec)
B.4     if m' = ⊥ or r = r_B mod 2 then
B.5       err ← 1                                      // error detected
B.6     else
B.7       T^B ← T^B ∘ b_rec
B.8       err ← 0
        end

        // Send a Message (even time-step)
B.9     if err = 0 then
B.10      b_send ← π(y | T^B)
B.11      T^B ← T^B ∘ b_send
B.12      r_B ← r_B + 1
B.13      m ← (b_send, r_B mod 2)
B.14    else                                           // err = 1: keep r_B and m unchanged.
B.15      send m
        end
      end
B.16  Output T^B
```

Proof. Consider Algorithm 2. It is immediate that r_B can increase by at most 1 in every round. In round i Bob starts by either appending $b_{rec} = b_A(i)$ to $\mathcal{T}^B(i)$ and setting $err = 0$; or keeping $\mathcal{T}^B(i)$ unchanged and setting $err = 1$. In the former case, Bob appends $b_{send} = b_B(i)$ to $\mathcal{T}^B(i) \circ b_A(i)$ and increases r_B. Otherwise, he keeps $\mathcal{T}^B(i)$ unchanged, and r_B remains the same as well.

In Algorithm 1 Alice may decrease her r_A, but note that she always begins round $i \leq t_A$ by increasing it (Line A.3) and appending $b_{send} = b_A(i)$ to $\mathcal{T}^A(i)$ (Line A.5). She then either decreases r_A back to what it was and in this case she erases $b_A(i)$ (see Lines A.12 and A.11), or she keeps it incremented and appends $b_{rec} = b_B(i)$ to $\mathcal{T}^A(i) \circ b_A(i)$. For $i > t_A$, i.e., after Alice terminates, the above trivially holds. ◀

▶ **Corollary 6.** *For any two rounds $i \leq j$, $\mathcal{T}^A(i) = \mathcal{T}^A(j)$ if and only if $r_A(i) = r_A(j)$. Similarly, $\mathcal{T}^B(i) = \mathcal{T}^B(j)$ if and only if $r_B(i) = r_B(j)$.*

Proof. Note that $\mathcal{T}^A, r_A$ are non-decreasing. Lemma 5 proves that these two variable increase simultaneously, which proves this corollary. The same holds for $\mathcal{T}^B, r_B$. ◀

▶ **Lemma 7.** *For $i \in [t_A + 1]$, one of the following conditions holds:*
1. *$r_B(i) = r_A(i)$ and $\mathcal{T}^B(i) = \mathcal{T}^A(i)$ or,*
2. *$r_B(i) = r_A(i) + 1$ and $\mathcal{T}^B(i) = \mathcal{T}^A(i) \circ b_A(i-1) \circ b_B(i-1)$.*

Furthermore, in either case, $\mathcal{T}^A$ and $\mathcal{T}^B$ are prefixes of $\mathcal{T}^\pi$.

Proof. We prove the lemma by induction on the round i. In the first round, $r_A(1) = r_B(1) = 0$ and $\mathcal{T}^B(1) = \mathcal{T}^A(1) = \lambda$ (see Lines A.1, B.1), thus Item 1 is satisfied. Note that $\mathcal{T}^A$ and $\mathcal{T}^B$ are trivially the prefixes of $\mathcal{T}^\pi$. Now assume that the conditions hold at (the beginning

of) round $i \leq t_A$ and consider what happens during this round. Note that both parties are active during round i.

Case 1: $r_A(i) = r_B(i)$. Suppose Alice receives an (uncorrupted) message from Bob that carries the parity $r_A(i) + 1$ (i.e., when Alice executes Line A.8 and not the else part of Lines A.11–A.12). This means that Bob sends a message with parity $r_A(i)+1 = r_B(i)+1$. Since $r_B(i) = r_A(i)$, this can only happen if Bob executed Lines B.13 and B.12, hence, $r_B(i+1) = r_B(i) + 1$. If Alice receives a corrupted message, she decrements r_A (that she had increased at the beginning of the round) and also deletes the last message from her transcript. Bob however may have received Alice's message correctly and in that event, he will increment his value of r_B and update his transcript.

Note that if $r_B(i+1) = r_A(i+1)$, then by the induction hypothesis and Lemma 5 both transcripts either remained the same in round i (so they are still the same at the beginning of round $i+1$), or both transcripts increased by appending $b_A(i) \circ b_B(i)$ to each, so they are still equal. Here, $b_A(i) = \pi(x \mid \mathcal{T}^A(i))$ and $b_B(i) = \pi(y \mid \mathcal{T}^B(i) \circ b_A(i))$. Similarly, if $r_B(i+1) = r_A(i+1) + 1$, Lemma 5 establishes that $\mathcal{T}^A(i)$ hasn't changed in round i, while $\mathcal{T}^B(i+1) = \mathcal{T}^B(i) \circ b_A(i) \circ b_B(i)$, which by the induction hypothesis gives Item 2. Since $b_A(i)$ and $b_B(i)$ are the correct continuations of $\mathcal{T}^A(i)$ from π, the above discussion proves that $\mathcal{T}^A(i+1)$ and $\mathcal{T}^B(i+1)$ are prefixes of $\mathcal{T}^\pi$.

Case 2: $r_B(i) = r_A(i) + 1$. In this case, whether Bob receives an erasure or an uncorrupted message, he sets $err = 1$. Indeed, if Alice's message is not erased, then the parity Bob receives equals his saved parity (since Alice holds $r_A(i) = r_B(i) - 1$ and she increases it by one in Line A.3 before sending it to Bob). In both cases Bob does not change r_B, i.e., $r_B(i+1) = r_B(i)$ and he sends the message $m_B(i) = (b_B, r_B \mod 2)$ (see Lines B.5 and B.15) from his memory. Consequently, Bob's transcript doesn't change in round i. So, $\mathcal{T}^B(i+1) = \mathcal{T}^B(i) = \mathcal{T}^A(i) \circ b_A(i-1) \circ b_B(i-1)$.

If Alice receives an erasure she sets $r_A(i+1)$ to be same as $r_A(i)$ (see Line A.12) and sets $\mathcal{T}^A(i+1) = \mathcal{T}^A(i)$. Since, $r_A(i+1) = r_A(i)$ and $r_B(i+1) = r_B(i)$ the claim holds. However, if Alice receives an uncorrupted message, she notices that $r_{rec} = (r_A(i) + 1) \bmod 2$ (see Line A.8) and she does not decrement r_A. In this case, $r_A(i+1) = r_A(i) + 1 = r_B(i) = r_B(i+1)$ and $\mathcal{T}^A(i+1) = \mathcal{T}^A(i) \circ b_A(i) \circ b_B(i)$. Now we prove that $b_A(i) = b_A(i-1)$ and $b_B(i) = b_B(i-1)$, thus in the former case, $\mathcal{T}^B(i+1) = \mathcal{T}^A(i+1) \circ b_A(i) \circ b_B(i)$ or $\mathcal{T}^A(i+1) = \mathcal{T}^B(i+1)$. in the latter. Since in round i, Bob sets $err = 1$ we have that $m_B(i) = m_B(i-1)$ whence $b_B(i) = b_B(i-1)$. To prove that the same holds for b_A we will need the following simple claim.

▷ Claim 8. $r_A(i) = r_A(i-1)$.

Proof. Supposing round $i-1$ satisfies Item 1, we must have that $r_A(i) = r_A(i-1)$. If this were not true, then $r_B(i) = r_A(i) + 1 = (r_A(i-1) + 1) + 1 = r_B(i-1) + 2$ which is a contradiction of Lemma 5. On the other hand, if round $i-1$ satisfies Item 2, then we know r_B cannot increase in round $i-1$, whence, $r_B(i) = r_B(i-1)$. Using this we have, $r_A(i-1) + 1 = r_B(i-1) = r_B(i) = r_A(i) + 1$. ◁

Following the above claim, $r_A(i-1) = r_A(i)$, and we get that $\mathcal{T}^A(i-1) = \mathcal{T}^A(i)$ (Lemma 5). Therefore, $b_A(i-1) = \pi(x \mid \mathcal{T}^A(i-1)) = \pi(x \mid \mathcal{T}^A(i)) = b_A(i)$. From the induction hypothesis, $\mathcal{T}^B(i+1) = \mathcal{T}^B(i)$ is a prefix of $\mathcal{T}^\pi$. From the above discussion, we know that either $\mathcal{T}^A(i+1) = \mathcal{T}^A(i) \circ b_A(i) \circ b_B(i) = \mathcal{T}^B(i+1)$ or $\mathcal{T}^A(i+1) = \mathcal{T}^A(i)$. In the former case, it is clear that $\mathcal{T}^A(i+1)$ is a prefix of $\mathcal{T}^\pi$ whereas in the latter case, since $\mathcal{T}^A(i)$ is a prefix of $\mathcal{T}^\pi$ (by the induction hypothesis) so is $\mathcal{T}^A(i+1)$. ◀

The following lemma implies that after Alice terminates, no matter what erasures Bob sees, his values of r_B and $\mathcal{T}^B$ do not change.

▶ **Lemma 9.** *For i such that $t_A < i \leq t_B$, round i satisfies Item 1 of Lemma 7 and $r_A(i) = \frac{N}{2}$.*

Proof. Recall that $t_A < t_B$. Since Alice exits in round t_A, it must hold that at the end of this round $r_A = \frac{N}{2}$, yet, $r_A(t_A) = \frac{N}{2} - 1$ for otherwise, Alice would have terminated in the end of round $t_A - 1$. Via Lemma 7 we know that $r_B(t_A) \in \{\frac{N}{2} - 1, \frac{N}{2}\}$. If $r_B(t_A) = \frac{N}{2} - 1$ we know from the proof of case 1 in Lemma 7 that since r_A increases in round t_A, r_B also increases. In the other case, namely, when $r_B(t_A) = \frac{N}{2}$, we know from the proof of Lemma 7 (case 2) that r_B does not change, i.e., $r_B(t_A + 1) = r_B(t_A) = \frac{N}{2}$.

After Alice exits, Bob can either hear silence or an erasure and therefore for all rounds $i > t_A$, Bob sets $err = 1$ and consequently $r_B(i)$ is never incremented. It follows that for all $i > t_A$, $r_B(i) = \frac{N}{2}$. The second part of the claim follows from Corollary 6 and Lemma 7, since $\mathcal{T}^A(t_A + 1) = \mathcal{T}^B(t_A + 1)$ and since r_A, r_B do not change anymore. ◀

We are now ready to prove the main theorem and show that we can correctly simulate the noiseless protocol π under the specifications of Theorem 1. We first state Lemma 10 which will be help us bound the communication and also show that Alice eventually terminates.

▶ **Lemma 10.** *In any round $i \in [t_A]$ where there are no erasures at all, $r_A(i+1) = r_A(i) + 1$.*

Proof. Note that both parties are still active in round i. Consider the two cases of Lemma 7. If $\mathcal{T}^A(i) = \mathcal{T}^B(i)$ and both messages of round i are not erased, we showed that r_A increases (case 1). Similarly, if $\mathcal{T}^B(i) = \mathcal{T}^A(i) \circ b_A(i-1) \circ b_B(i-1)$ and no erasures occur, Alice extends her transcript and r_A increases again (case 2). ◀

▶ **Theorem 11.** *Let π be an alternating binary protocol and $T \in \mathbb{N}$ be an arbitrary integer. There exists a coding scheme Π over a 4-ary alphabet such that for any instance of Π that suffers at most T erasures overall, Alice and Bob both output $\mathcal{T}^\pi$. The simulation Π communicates at most $\mathsf{CC}(\pi) + 2T$ symbols, and has $\mathsf{CC}(\Pi) \leq 2\mathsf{CC}(\pi) + 4T$.*

Proof. Lemma 7 guarantees that at every given round, Alice and Bob hold a correct prefix of $\mathcal{T}^\pi$. Moreover, we know that by the time Alice terminates, her transcript (and hence Bob's transcript) is of length at least N, which follows from Lemma 9 and Lemma 5, i.e., from the fact that $r_A = N/2$ at termination, and that every time r_A increases by one, the length of $\mathcal{T}^A$ increases by two. Finally, note that if the number of erasures is bounded by T, then Alice will eventually reach termination because, after T erasures, r_A increases by one in every round (Lemma 10), until it reaches $N/2$ and Alice terminates.

Finally, we need to prove that the communication behaves as stated. Assume T' erasures happen up to round t_A and $T'' = T - T'$ erasures happen in rounds $[t_A + 1, t_B]$. Since every round without erasures advances r_A by one (again from Lemma 10), and since when Alice terminates we have $r_A = N/2$ (Lemma 9), then $t_A \leq T' + N/2$. Furthermore, after Alice terminates it takes one unerased (odd) round to make Bob terminate as well. Hence, at every round after t_A and until Bob terminates, Alice's silence must be erased. It follows that $t_B - t_A \leq 1 + T''$. Thus, $t_B \leq 1 + T'' + t_A \leq 1 + T + N/2$.

Every round $i \in [t_A]$ consists of two transmissions, while every round $t_A < i < t_B$ contains only a single transmission–Bob's transmission, excluding round t_B where Bob hears silence and terminates without sending a message. The total number of transmissions is then,

$$t_A + t_B - 1 \leq N + 2T' + T'' \leq N + 2T.$$

Recall that $|\Sigma| = 4$, hence $\mathsf{CC}(\Pi) \leq (N + 2T)\log 4 = 2(\mathsf{CC}(\pi) + 2T)$. ◀

3.3 Noise Resilience and Code Rate

We can compare the above result to the case where the noise is bounded as a fraction of the symbols communicated in the protocol [30, 14, 12]. In our setting, the noise amount can be arbitrary. In order to compare it to the bounded-noise model, we ask the following question. Assume an instance of Π with large amount of noise T ($T \gg N$). Then, what fraction does the noise make out of the entire communication.

As a corollary of Theorem 11 it is easy to see that the fraction of noise is lower bounded by $\frac{T}{N+2T}$ whose limit, as T tends to infinity, is $1/2$. Indeed, $1/2$ is an upper bound on the noise fraction in the bounded-noise setting [14].[4] Furthermore, if the noise is bounded to be a δ-fraction of the total communication, for some $\delta < 1/2$ then, $T \leq \delta \cdot 2t_B \leq \delta(N + 2T)$ and so $T \leq \frac{\delta N}{1-2\delta}$. This implies a maximal asymptotic code rate of $1/2$. Indeed,

$$R = \frac{\mathsf{CC}(\pi)}{\mathsf{CC}(\Pi)} \geq \frac{1}{\log|\Sigma|} \cdot \frac{N}{N+2T} \geq \frac{1}{2} \cdot \frac{N}{N + 2\frac{\delta N}{1-2\delta}} \geq \frac{1}{2}(1-2\delta) = \frac{1}{2} - \delta.$$

As T is unbounded relative to $\mathsf{CC}(\pi)$ and we can potentially get a zero rate, a more interesting measure is the "waste" factor, i.e., how much the communication of Π increases per single noise, for large T. In our scheme it is easy to see that each corruption delays the simulation by one round, that is, it wastes two symbols (4 bits). This implies a waste factor of 4 bits per corruption, $\omega = \lim_{T\to\infty} \frac{\mathsf{CC}(\Pi)}{T} = \frac{2N+4T}{T} = 4$.

Finally, we mention that our result extends to binary alphabet by naively encoding each symbols as two bits (this also proves the second part of Theorem 1). However, this results in a reduced tolerable noise rate of $\frac{1}{4}$. Similar to the scheme in [12], the noise resilience can be improved to $\frac{1}{3}$ by encoding each symbol via an error correcting code of cardinality 4 and distance $\frac{2}{3}$, e.g., {000,011,110,101}. In this case, two bits must be erased in order to invalidate a round. Each round (two timesteps) consists the transmission of six bits. Hence, the obtained resilience is $2/6 = 1/3$, similar to the best known resilience in the bounded-noise setting with binary alphabet [12].

References

1. Abhinav Aggarwal, Varsha Dani, Thomas P. Hayes, and Jared Saia. Distributed Computing with Channel Noise. Cryptology ePrint Archive, Report 2017/710, 2017. URL: `https://eprint.iacr.org/2017/710`.
2. Abhinav Aggarwal, Varsha Dani, Thomas P. Hayes, and Jared Saia. Sending a Message with Unknown Noise. In *Proceedings of the 19th International Conference on Distributed Computing and Networking*, ICDCN '18, pages 8:1–8:10, 2018. `doi:10.1145/3154273.3154318`.
3. Shweta Agrawal, Ran Gelles, and Amit Sahai. Adaptive Protocols for Interactive Communication. In *2016 IEEE International Symposium on Information Theory (ISIT)*, 2016. `doi:10.1109/ISIT.2016.7541368`.
4. Zvika Brakerski, Yael T. Kalai, and Moni Naor. Fast Interactive Coding Against Adversarial Noise. *J. ACM*, 61(6):35:1–35:30, December 2014. `doi:10.1145/2661628`.
5. G. Brassard, A. Nayak, A. Tapp, D. Touchette, and F. Unger. Noisy Interactive Quantum Communication. In *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, pages 296–305, October 2014. `doi:10.1109/FOCS.2014.39`.

[4] Note that a noise level of $1/2$ is also achievable for interactive coding over erasure channels in the bounded-noise setting, for alphabets of size at least 4 [12, 14]. The maximal noise for erasure channels with *binary* or *ternary* alphabet is still open.

6 M. Braverman and K. Efremenko. List and Unique Coding for Interactive Communication in the Presence of Adversarial Noise. *SIAM Journal on Computing*, 46(1):388–428, 2017. doi:10.1137/141002001.

7 M. Braverman, R. Gelles, J. Mao, and R. Ostrovsky. Coding for Interactive Communication Correcting Insertions and Deletions. *IEEE Transactions on Information Theory*, 63(10):6256–6270, October 2017. doi:10.1109/TIT.2017.2734881.

8 Mark Braverman. Towards deterministic tree code constructions. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, pages 161–167. ACM, 2012. doi:10.1145/2090236.2090250.

9 Mark Braverman and Anup Rao. Toward Coding for Maximum Errors in Interactive Communication. *Information Theory, IEEE Transactions on*, 60(11):7248–7255, 2014. doi:10.1109/TIT.2014.2353994.

10 Ronald Cramer, Yevgeniy Dodis, Serge Fehr, Carles Padró, and Daniel Wichs. Detection of Algebraic Manipulation with Applications to Robust Secret Sharing and Fuzzy Extractors. In *Advances in Cryptology – EUROCRYPT 2008*, pages 471–488, 2008. doi:10.1007/978-3-540-78967-3_27.

11 Varsha Dani, Thomas P. Hayes, Mahnush Movahedi, Jared Saia, and Maxwell Young. Interactive communication with unknown noise rate. *Information and computation*, 261:464–486, 2018. doi:10.1016/j.ic.2018.02.018.

12 Klim Efremenko, Ran Gelles, and Bernhard Haeupler. Maximal Noise in Interactive Communication Over Erasure Channels and Channels With Feedback. *IEEE Trans. Information Theory*, 62(8):4575–4588, 2016. doi:10.1109/TIT.2016.2582176.

13 Klim Efremenko, Elad Haramaty, and Yael Kalai. Interactive Coding with Constant Round and Communication Blowup. *Electronic Colloquium on Computational Complexity (ECCC)*, 25:54, 2018. URL: https://eccc.weizmann.ac.il/report/2018/054.

14 Matthew Franklin, Ran Gelles, Rafail Ostrovsky, and Leonard J. Schulman. Optimal Coding for Streaming Authentication and Interactive Communication. *Information Theory, IEEE Transactions on*, 61(1):133–145, January 2015. doi:10.1109/TIT.2014.2367094.

15 R. Gelles and B. Haeupler. Capacity of Interactive Communication over Erasure Channels and Channels with Feedback. *SIAM Journal on Computing*, 46(4):1449–1472, 2017. doi:10.1137/15M1052202.

16 R. Gelles, B. Haeupler, G. Kol, N. Ron-Zewi, and A. Wigderson. Explicit Capacity Approaching Coding for Interactive Communication. *IEEE Transactions on Information Theory*, 64(10):6546–6560, October 2018. doi:10.1109/TIT.2018.2829764.

17 Ran Gelles. Coding for Interactive Communication: A Survey. *Foundations and Trends® in Theoretical Computer Science*, 13(1–2):1–157, 2017. doi:10.1561/0400000079.

18 Ran Gelles and Siddharth Iyer. Interactive coding resilient to an unknown number of erasures. *CoRR*, abs/1811.02527, 2018. A preliminary full version of this paper. arXiv:1811.02527.

19 Ran Gelles, Ankur Moitra, and Amit Sahai. Efficient Coding for Interactive Communication. *Information Theory, IEEE Transactions on*, 60(3):1899–1913, March 2014. doi:10.1109/TIT.2013.2294186.

20 Mohsen Ghaffari and Bernhard Haeupler. Optimal Error Rates for Interactive Coding II: Efficiency and List Decoding. In *Foundations of Computer Science (FOCS), IEEE 55th Annual Symposium on*, pages 394–403, 2014.

21 Mohsen Ghaffari, Bernhard Haeupler, and Madhu Sudan. Optimal Error Rates for Interactive Coding I: Adaptivity and Other Settings. In *STOC '14: Proceedings of the 46th Annual ACM Symposium on Theory of Computing*, pages 794–803, 2014. doi:10.1145/2591796.2591872.

22 Bernhard Haeupler. Interactive Channel Capacity Revisited. In *Foundations of Computer Science (FOCS), IEEE 55th Annual Symposium on*, pages 226–235, 2014.

23 Bernhard Haeupler, Amirbehshad Shahrasbi, and Ellen Vitercik. Synchronization Strings: Channel Simulations and Interactive Coding for Insertions and Deletions. In *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*, volume 107 of *Leibniz*

International Proceedings in Informatics (LIPIcs), pages 75:1–75:14, 2018. doi:10.4230/LIPIcs.ICALP.2018.75.

24 Joseph Y. Halpern and Yoram Moses. Knowledge and Common Knowledge in a Distributed Environment. *J. ACM*, 37(3):549–587, July 1990. doi:10.1145/79147.79161.

25 IEEE. IEEE standard for Ethernet. *IEEE Std 802.3-2015 (Revision of IEEE Std 802.3-2012)*, pages 1–4017, March 2016. doi:10.1109/IEEESTD.2016.7428776.

26 Gillat Kol and Ran Raz. Interactive channel capacity. In *STOC '13: Proceedings of the 45th annual ACM Symposium on theory of computing*, pages 715–724, 2013. doi:10.1145/2488608.2488699.

27 Debbie Leung, Ashwin Nayak, Ala Shayeghi, Dave Touchette, Penghui Yao, and Nengkun Yu. Capacity Approaching Coding for Low Noise Interactive Quantum Communication. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2018, pages 339–352, 2018. doi:10.1145/3188745.3188908.

28 Denis Pankratov. On the Power of Feedback in Interactive Channels. [Online:] http://people.cs.uchicago.edu/~pankratov/papers/feedback.pdf, 2013.

29 Leonard J. Schulman. Communication on noisy channels: a coding theorem for computation. *Foundations of Computer Science, Annual IEEE Symposium on*, pages 724–733, 1992. doi:10.1109/SFCS.1992.267778.

30 Leonard J. Schulman. Coding for interactive communication. *IEEE Transactions on Information Theory*, 42(6):1745–1756, 1996. doi:10.1109/18.556671.

31 Leonard J. Schulman. A postcript to "Coding for Interactive Communication". [Online:] http://www.cs.caltech.edu/~schulman/Papers/intercodingpostscript.txt, 2003.

32 Alexander A. Sherstov and Pei Wu. Optimal Interactive Coding for Insertions, Deletions, and Substitutions. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 240–251, October 2017. doi:10.1109/FOCS.2017.30.

A Generic Undo Support for State-Based CRDTs

Weihai Yu
University of Tromsø – The Arctic University of Norway, Norway
weihai.yu@uit.no

Victorien Elvinger
Université de Lorraine, CNRS, Inria, LORIA, F-54000, Nancy, France
victorien.elvinger@inria.fr

Claudia-Lavinia Ignat
Université de Lorraine, CNRS, Inria, LORIA, F-54000, Nancy, France
claudia.ignat@inria.fr

Abstract

CRDTs (Conflict-free Replicated Data Types) have properties desirable for large-scale distributed systems with variable network latency or transient partitions. With CRDT, data are always available for local updates and data states converge when the replicas have incorporated the same updates. Undo is useful for correcting human mistakes and for restoring system-wide invariant violated due to long delays or network partitions.

There is currently no generally applicable undo support for CRDTs. There are at least two reasons for this. First, there is currently no abstraction that we can practically use to capture the relations between undo and normal operations with respect to concurrency and causality. Second, using inverse operations as the existing partial solutions, the CRDT designer has to hard-code certain rules and design a new CRDT for almost every operation that needs undo support.

In this paper, we present an approach to generic support of undo for CRDTs. The approach consists of two major parts. We first work out an abstraction that captures the semantics of concurrent undo and redo operations through equivalence classes. The abstraction is a natural extension of undo and redo in sequential applications and is straightforward to implement in practice. By using this abstraction, we then device a mechanism to augment existing CRDTs. The mechanism provides an "out of the box" support for undo without the involvement of the CRDT designers. We also present a practical application of the approach in collaborative editing.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Computing methodologies → Concurrent algorithms; Information systems → Data replication tools; Human-centered computing → Synchronous editors; Human-centered computing → Asynchronous editors

Keywords and phrases Data replication, eventual consistency, state-based CRDT, delta-state CRDT, concurrent undo

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.14

Funding The research was mostly carried out during Yu's sabbatical stay at Inria, LORIA, financed by University of Tromsø – The Arctic University of Norway.

Acknowledgements The authors thank the members of the COAST team at Inria, LORIA, in particular Matthieu Nicolas and Gérald Oster, for interesting discussions.

1 Introduction

The CAP theorem ([11, 14]) states that in a networked system, it is impossible to simultaneously ensure all three desirable properties, namely (C) consistency equivalent to a single up-to-date copy of data, (A) availability of that data for update and (P) tolerance to network partition. [7] revisited the theorem and clarified some common misunderstandings. Among these, the three properties are continuous rather than binary and partition is a function of

© Weihai Yu, Victorien Elvinger, and Claudia-Lavinia Ignat;
licensed under Creative Commons License CC-BY
23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller; Article No. 14; pp. 14:1–14:17
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

latency. [7] also referred to two useful approaches to partition management that are highly relevant to the work we are presenting in this paper: (1) CRDT, with which replicated data provably converge after a partition (delay) and (2) compensation (undo), which can be used to fix violation of global invariant during partition.

CRDTs [23], or Conflict-free Replicated Data Types, have properties desirable for large-scale distributed applications. A site independently updates its local replica (i.e. a site is always available for update). The states of replicas converge when they have incorporated the same set of updates (referred to as *strong convergence* in [23]). Applications adopting CRDTs include distributed key-value stores [8] and collaborative editors ([17, 29, 20, 26, 28]). There has also been active research on CRDT-based transaction processing ([2, 15, 19]).

During a network partition or after long network delay, systems' global invariant could be violated, such as overbooking of resources or pre-mature commit of sub-transactions [12]. For applications such as online shopping and collaborative editing, human users introduce additional delay and mistakes. Undo (or compensation [12]) is a generic tool to fix human mistakes or restore global invariant.

Currently, there is no generic support of undo for CRDTs. There are at least two reasons for this. First, there is currently no simple applicable abstraction that sufficiently captures the relations between undo and normal operations with respect to concurrency and causality. Second, using inverse operations as the existing partial solutions, the CRDT designer has to hard-code certain rules and design a new CRDT for almost every operation that needs undo support. We explain these issues in detail in later sections §5.1 and §4.3.

Our first contribution is an abstraction that defines the semantics of concurrent undo and redo operations through equivalence classes. The abstraction correctly captures concurrency and causality of undo and redo operations. It is a natural extension to undo and redo of sequential systems and hence is easy to understand and straightforward to implement in practice. The abstraction applies generally beyond the context of CRDTs.

Our second contribution is a generic approach to augmenting existing CRDTs with an "out of the box" support for undo. Unlike the current partial solutions where the CRDT designer has to design a new CRDT for nearly every inverse operation, with our approach, the CRDT designer does not have to get involved in the design of individual inverse operations.

The original CRDT paper [23] presented two families of CRDT approaches, namely state-based and operation-based. There have been improvement and refinement on both approaches, the most representative being [3] on state-based and [5] on operation-based. Our work focuses on state-based CRDTs and is based on [3].

The paper is organized as follows. §2 describes the model of the systems our work applies to and §3 presents the notations we use. §4 reviews the background of CRDTs. §5 presents our first main contribution, the abstraction for concurrent undo and redo operations. §6 describes our second main contribution, to generically support undo for existing CRDTs. §7 shows a practical application of our work in collaborative editing. §8 discusses related work. §9 concludes.

2 System Model

A distributed system consists of sites with globally unique identifiers. We use $\mathbb{I}$ for the set of site identifiers. Sites do not share memory. They maintain durable states. Sites may crash, but will eventually recover to the durable state at the time of the last crash.

A site can send messages to any other site in the system through an asynchronous and unreliable network. There is no upper bound on message delay. The network may discard,

reorder or duplicate messages, but it cannot corrupt messages. Through re-sending, messages will eventually be delivered. The implication is that there can be network partitions, but disconnected sites will eventually get connected.

3 Notations

$\mathbb{N}$ is the set of natural numbers. $\mathbb{B}$ is the set of Boolean values. $\mathbb{B} = \{\mathsf{False}, \mathsf{True}\}$. $\mathcal{P}(S)$ denotes the power set on S. Most sets in this paper are partially ordered and have a least element $\perp$ (also known as the bottom element).

Set comprehension is of the form $\{x \in S | \ \mathsf{pred}(x)\}$ or $\{f(x) | x \in S\}$, where f is a function and pred is a predicate.

We use $m\colon K \hookrightarrow V$ to denote a partial function where $\mathsf{dom}(m) \subseteq K$. A partial function can be represented as a set of pairs $\{\langle k, m(k)\rangle | k \in \mathsf{dom}(m)\}$. When $k \in K \wedge k \notin \mathsf{dom}(m)$ and V has a bottom $\perp_V$, we use $m(k) = \perp_V$ for convenience. For example, given a partial function $p\colon \mathbb{N} \hookrightarrow \mathbb{N}$ and $\mathsf{dom}(p) = \varnothing$, we use $p(n) = 0$ for any $n \in \mathbb{N}$, because $\perp_{\mathbb{N}} = 0$. Due to this convenience, we do not need an initialization $p(n) = 0$ as in the case of a total function.

The notation $m\{k \mapsto v\}$ represents an update of the function m for a new value v associated with the key k.

The notation $\mathsf{f}(x)$ is like a function or procedure in a conventional programming language. In this paper, it can be a query, a mutator (an operation) or a predicate. We may write $\mathsf{f}_y(x)$ for $\mathsf{f}(x, y)$ to make the signatures of functions look consistent in different contexts. For example, $\mathsf{inc}(x)$ increments a counter x, while $\mathsf{inc}(x, A)$, or better $\mathsf{inc}_A(x)$, increments a counter x at site A.

4 CRDT Background

A CRDT is a data type specifically designed for data replicated at different sites. A site queries and updates its local replica independently (i.e. without coordination with other sites). The data is always available for update, but the data states at different sites may diverge. From time to time, the sites send their updates asynchronously to other sites with an anti-entropy protocol. To incorporate the updates made at the other sites, a site merges the received updates with its local replica. A CRDT has the property that when all sites have incorporated the same set of updates, the replicas converge.

There are two families of CRDT approaches, namely operation-based and state-based. For an operation-based CRDT [23], a message for an update is an encoding of the operation that made the corresponding update. A site that receives the message runs a special procedure to incorporate the update. To enforce convergence, the operations of an operation-based CRDT should commute, i.e. the executions of the same set of operations in different orders should have the same effect. A CRDT is *purely* operation-based if the encoding and incorporation of operations are trivial, in the sense that they are independent of the state at which the operation is performed [5]. Pure operation-based CRDTs require reliable causal delivery of messages.

For a state-based CRDT, as originally presented in [23], a message for updates is the data state of the replica in its entirety. The site that receives the message incorporates the updates by merging the received state and its local state. When the possible states of the data form a join-semilattice (see §4.1 below), the merge is the join of the two states. Convergence is implied by the join-semilattice.

$$\mathsf{GSet}(E) \stackrel{\text{def}}{=} \mathcal{P}(E)$$
$$\mathsf{add}(e, s) \stackrel{\text{def}}{=} \{e\} \cup s$$
$$\mathsf{add}^{\delta}(e, s) \stackrel{\text{def}}{=} \begin{cases} \{e\} & \text{if } e \notin s \\ \bot & \text{otherwise} \end{cases}$$
$$s \sqcup s' \stackrel{\text{def}}{=} s \cup s'$$
$$\mathsf{in}(e, s) \stackrel{\text{def}}{=} e \in s$$

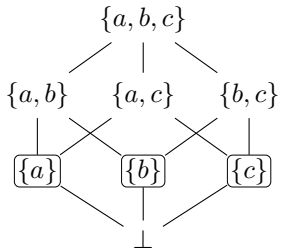


Figure 1 GSet CRDT and Hasse diagram of states.

As our work focuses on state-based CRDTs, in the following subsections, we present the main theory underlying this family of CRDTs, including delta-state CRDTs [3], which improve the original state-based CRDTs. We also discuss a typical design of inverse operations in state-based CRDTs and why this is usually not sufficient as a mechanism of undo.

4.1 State-based CRDTs

A *state-based CRDT* is a tuple $\langle \mathcal{S}, \sqsubseteq, s^0, \mathcal{Q}, \mathcal{M}, \sqcup \rangle$, where $\mathcal{S}$ is a poset of states under partial order $\sqsubseteq$, $s^0 \in \mathcal{S}$ is an initial state, $\mathcal{Q}$ is a set of queries on the states, $\mathcal{M}$ is a set of mutators for performing updates on the states, and $\sqcup$ is a join operation on states. Furthermore, the state poset with $\sqcup$ is a join-semilattice. In this paper, we use the term *operation* as a particular instance of state update defined by a mutator. For example, $m \in \mathcal{M}$ is a mutator, whereas $m(s)$ is a state update, hence an operation. Consequently, for two different states s_1 and s_2, $m(s_1)$ and $m(s_2)$ are two different operations.

For a poset P under the partial order $\sqsubseteq$, a join operation $x \sqcup y$ returns the least upper bound of elements x and y in P. The join operation is idempotent, commutative and associative. The poset P is a *join-semilattice* iff $x \sqcup y$ exists for any x and y in P [13]. Some join-semilattices have a least element $\bot$, also known as the bottom element. A power set, under the partial order of set inclusion $\subseteq$ and with set union $\cup$ as join, is a classic example of a join-semilattice that has a bottom element $\bot = \varnothing$. For every CRDT discussed in this paper, we assume a bottom state $\bot$.

For a state-based CRDT, every state update is an inflation. That is, for any mutator $m \in \mathcal{M}$ and state $s \in \mathcal{S}$, $s \sqsubseteq m(s)$. When a local state s merges with a received remote state s', the new local state becomes $s \sqcup s'$. Because local updates are inflations and merges are the results of joins, at each site, state updates are monotonic under $\sqsubseteq$. In other words, every new state s_{n+1} subsumes a previous state s_n, i.e. $s_n \sqsubseteq s_{n+1}$ for any $n \geq 0$.

Figure 1 (left) shows GSet, a state-based CRDT for grow-only sets, where $\mathcal{S} \stackrel{\text{def}}{=} \mathcal{P}(E)$ for a set E of possible elements, $\sqsubseteq \stackrel{\text{def}}{=} \subseteq$, $s^0 \stackrel{\text{def}}{=} \varnothing$, $\mathcal{Q} \stackrel{\text{def}}{=} \{\mathsf{in}\}$, $\mathcal{M} \stackrel{\text{def}}{=} \{\mathsf{add}\}$ and $\sqcup \stackrel{\text{def}}{=} \cup$. (The figure also shows a delta-mutator add^{δ} that will be explained in §4.2.) Obviously, an update through $\mathsf{add}(e)$ is an inflation, because $s \subseteq \{e\} \cup s$. Figure 1 (right) shows the Hasse diagram of the states in a GSet. A Hasse diagram shows only the "direct links" between states (known as the *cover* relation $\sqsubset_c$ [13]).

GSet is an example of an *anonymous* CRDT. A CRDT is anonymous if its operations are not specific to the sites that perform the operations and hence do not refer to site identifiers. Two sites can concurrently perform the operations defined by the same mutator. We say that these two sites perform *the same anonymous operations* concurrently. For example, when site A performs operation $\mathsf{add}(a, s_1)$ and site B concurrently performs operation $\mathsf{add}(a, s_2)$, the sites perform the same anonymous operation $\mathsf{add}(a)$.

$$\mathsf{GCounter} \stackrel{\text{def}}{=} \mathbb{I} \hookrightarrow \mathbb{N}$$

$$\mathsf{inc}_i(s) \stackrel{\text{def}}{=} s\{i \mapsto s(i)+1\}$$

$$\mathsf{inc}_i^\delta(s) \stackrel{\text{def}}{=} \{\langle i, s(i)+1\rangle\}$$

$$s \sqcup s' \stackrel{\text{def}}{=} \{\langle i, \max\big(s(i), s'(i)\big)\rangle \mid i \in \mathsf{dom}(s) \cup \mathsf{dom}(s')\}$$

$$\mathsf{value}(s) \stackrel{\text{def}}{=} \textstyle\sum_{i \in \mathsf{dom}(s)} s(i)$$

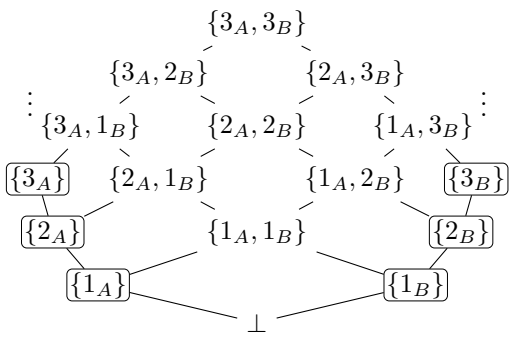


Figure 2 GCounter CRDT and Hasse diagram of states.

On the other hand, a CRDT is *named* if a site can only update the part of the state that is specific to that site. Different sites cannot perform the same operation concurrently. Figure 2 (left) shows GCounter, a state-based CRDT for grow-only counters. It uses a partial function (or a key-value map) $\mathbb{I} \hookrightarrow \mathbb{N}$ to simulate a globally replicated counter. The sites update the key-value map similar to a version vector [18]. When site i increments the counter using operation inc_i, only the value mapped from the key i gets incremented. GCounter is named because operation inc_i is specific to site i and only site i can perform it. Figure 2 (right) shows the Hasse diagram of the states in a GCounter. In the figure, 2_A denotes the pair $\langle A, 2\rangle$, to expose the meaning "value 2 at site A".

4.2 Delta-state CRDTs

Using state-based CRDTs, as originally presented, is costly in practice, because states in their entirety are sent as messages. Delta-state CRDTs address this issue [3]. They are based on the concept of join-irreducible states.

An element x is *join-irreducible* in a poset P if it cannot be expressed as a join of other elements in P [13]. Formally, x is join-irreducible in P if $\forall y, z \in P : x = y \sqcup z \Rightarrow x = y \vee x = z$. We use $\mathcal{J}(P)$ for the set of join-irreducible elements of P.

For a finite join-semilattice, join-irreducible elements are those that have only one link below in the Hasse diagram [13]. In Figure 1 and Figure 2, the states in boxes are join-irreducible. The set of join-irreducible states of GSet, $\mathcal{J}(\mathcal{P}(E))$, consists of singleton sets. The set of join-irreducible states of GCounter, $\mathcal{J}(\mathbb{I} \to \mathbb{N})$ consists of singleton pair sets.

An important property of join-irreducible elements is that every element in a finite poset can be represented as a join of some join-irreducible elements. More precisely, given a finite poset P, for any $x \in P$, $x = \bigsqcup\{y \in \mathcal{J}(P) | y \sqsubseteq x\}$.

A delta-state CRDT has a delta-mutator m^δ for every mutator m of the corresponding state-based CRDT. Instead of returning the new updated state $m(s)$, m^δ returns a delta representation consisting only of join-irreducible states. The delta representation has the property $m(s) = s \sqcup m^\delta(s)$. For example, in Figure 1, add^δ is the delta counterpart of add. While $\mathsf{add}(e, s)$ returns the whole new state $\{e\} \cup s$, $\mathsf{add}^\delta(e, s)$ returns only a singleton set $\{e\}$ (when e was not in s and the mutation is effectively executed).

Now, instead of sending the whole state $m(s)$, a site only sends the delta representation $m^\delta(s)$, which is typically much smaller in size than $m(s)$. If a remote site has already incorporated s, a merge with $m^\delta(s)$ gives the same result as a merge with $m(s)$. We can thereby regard $m^\delta(s)$ as $m(s)$ where redundancy in s is eliminated.

$$
\begin{aligned}
2\mathsf{P}_{\mathbb{B}}\mathsf{Set}(E) &\stackrel{\text{def}}{=} E \hookrightarrow \mathbb{B} \\
\mathsf{add}(e, s) &\stackrel{\text{def}}{=} \{\langle e, \mathsf{False}\rangle\} \sqcup s \\
\mathsf{add}^{\delta}(e, s) &\stackrel{\text{def}}{=} \begin{cases} \{\langle e, \mathsf{False}\rangle\} & \text{if } e \notin \mathsf{dom}(s) \\ \bot & \text{otherwise} \end{cases} \\
\mathsf{remove}(e, s) &\stackrel{\text{def}}{=} \{\langle e, \mathsf{True}\rangle\} \sqcup s \\
\mathsf{remove}^{\delta}(e, s) &\stackrel{\text{def}}{=} \begin{cases} \{\langle e, \mathsf{True}\rangle\} & \text{if } \langle e, \mathsf{True}\rangle \notin s \\ \bot & \text{otherwise} \end{cases} \\
s \sqcup s' &\stackrel{\text{def}}{=} \{\langle e, s(e) \vee s'(e)\rangle | e \in \mathsf{dom}(s) \cup \mathsf{dom}(s')\} \\
\mathsf{in}(e, s) &\stackrel{\text{def}}{=} e \in \mathsf{dom}(s) \wedge s(e) = \mathsf{False}
\end{aligned}
$$

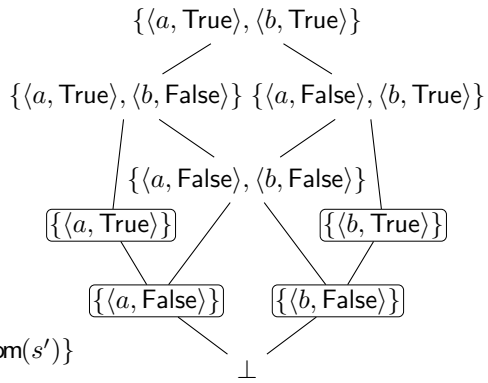


Figure 3 $2\mathsf{P}_{\mathbb{B}}\mathsf{Set}$ CRDT and Hasse diagram of states.

Because the delta representation is not an inflation, the anti-entropy protocol must do some extra work to achieve certain degree of causality [3]. Otherwise, the replicas will still eventually converge, but the sites may observe states out of causal order. In this paper, we focus on the design aspect of CRDTs and their undo support, and will not discuss the anti-entropy protocols.

4.3 Inverse operation as undo

Sometimes we may want to perform an inverse of an earlier update, for example, to remove an element that was earlier added into a set. Because updates in state-based CRDTs must be inflationary (§4.1), it is relatively easy to design CRDTs for those applications where the data grow in nature, such as grow-only set and grow-only counter. To support operations that make data shrink, such as the inverse operation of inflationary operations, we have to design new CRDTs using some special techniques. For example, we can keep the removed data as a kind of tombstones and let the queries achieve the shrinking effect. [23] and [3] presented different set CRDTs that have both add and remove operations.

Figure 3 (left) shows a set CRDT $2\mathsf{P}_{\mathbb{B}}\mathsf{Set}$ (two-phase set using Boolean flags) that is a variation of u-set in [23] and two-phase set $2\mathsf{PSet}$ in [3]. We associate every element added to the set with a Boolean flag indicating whether the element has been removed. More precisely, the states are a partial function $E \hookrightarrow \mathbb{B}$. We use pair $\langle e, \mathsf{False}\rangle$ when element e is added and $\langle e, \mathsf{True}\rangle$ when element e is removed. We adopt the conventional order of Boolean values $\mathsf{False} \sqsubset \mathsf{True}$. Hence, when an element is added and removed, the removal wins. (Note in the definitions of remove and $\sqcup$, $s(e) = \mathsf{False}$ when $e \notin \mathsf{dom}(s)$.) Figure 3 (right) shows the Hasse diagram of the states in $2\mathsf{P}_{\mathbb{B}}\mathsf{Set}$. For example, when state $\{\langle a, \mathsf{True}\rangle\}$ (i.e. element a has been removed) merges with state $\{\langle a, \mathsf{False}\rangle, \langle b, \mathsf{False}\rangle\}$ (i.e. both elements a and b are in the set), the new state is $\{\langle a, \mathsf{True}\rangle, \langle b, \mathsf{False}\rangle\}$ (i.e. only element b is in the set).

Using operation remove as an inverse operation of add in $2\mathsf{P}_{\mathbb{B}}\mathsf{Set}$ has a problem. The remove operation itself does not have an inverse operation. Once an element has been removed, it cannot be added back again. Actually, this problem is common among many CRDTs that provide some kind of inverse operations.

Causal CRDTs [3] such as OR-Set (observed-remove set [6], [16]) address this problem by associating state elements with causal contexts. A causal context is a set of event identifiers (typically a pair of a site identifier and a site-specific sequence number). Using causal contexts, we are able to tell explicitly which additions of an element have been later removed. Because

there is no upper bound on causal contexts, we can inverse any given (undo or redo) operation by inflation of associated causal contexts. However, maintaining causal contexts for every element can be costly, even though it is possible to compress causal contexts into vector states, especially under causal consistency. In our first contribution (§5), we work out an abstraction that allows us to use a single number as the smallest context without upper bound.

In general, inverse operations must be specially designed for the given operations and the design is normally not directly applicable to other operations or CRDTs. In our second contribution (§6), we present how to support undo in any state-based CRDT through a generic state transformation in the join semilattice space of the CRDT states.

5 Concurrent Undo and Redo Operations

This section presents our first main contribution. We formally characterize the concurrency and causality of undo and redo operations using equivalence classes. We can then represent the equivalence classes with single numbers called undo lengths. The abstraction presented in this section applies generally beyond the context of CRDTs.

5.1 Problem statement

The basic question is: when a site sees a set of undo and redo operations of an original normal operation *op*, should the site undo or redo *op*?

▶ **Example 1.** Site S_1 inserts an element e into a set with operation add_1, undoes the addition with undo_1 and then redoes it with redo_1. Site S_2 receives add_1, undoes it with undo_2, and then receives and integrates undo_1 and redo_1. Is element e in the set at site S_2? The answer should be "yes", because the concurrent undo_1 and undo_2 operations have the same intention, and redo_1, whose intention is to redo the effect of add_1, supersedes both.

In a sequential system, such as a single-user editor, undo and redo of the same normal operation happen in turn. We could simply count the length of the undo-redo chain. If the length is an odd number, the original operation is undone, otherwise, it is redone. In the example, site S_1 alone is like a sequential system. The length of the undo-redo chain at site S_1 is two and the addition of e should be redone.

Undo in concurrent applications has been an active research topic for decades, particularly in the area of collaborative editing ([1, 10, 29, 21, 22, 24, 25, 27, 28]). However, most of the published work does not account for concurrent undo and redo operations correctly.

Some of the latest work also counted the number of undo and redo operations to decide whether an original operation is finally undone or redone, but the result is unsatisfactory.

The approach presented in [25] counts the number of times an operation has been undone or redone. If it is an odd number, the original operation in undone. In the example, the number is 3 at site S_2, so the addition operation add_1 is incorrectly undone.

The approach reported in [27] counts the numbers of undo and redo operations separately. The undo or redo with the higher number wins. In the example, there are two undos and one redo at site S_2. Therefore undo wins and add_1 is incorrectly undone.

The root problem with these earlier approaches is that they do not define the semantics undo and redo operations with respect to concurrency and causality of the operations. In the example, the two concurrent undo operations undo_1 and undo_2 are both meant to undo the same operation add_1. Therefore they should have the same effect as a single undo. On the other hand, redo_1 happens causally after undo_1 (which is effectively the same as undo_2) and hence should have the final effect at site S_2.

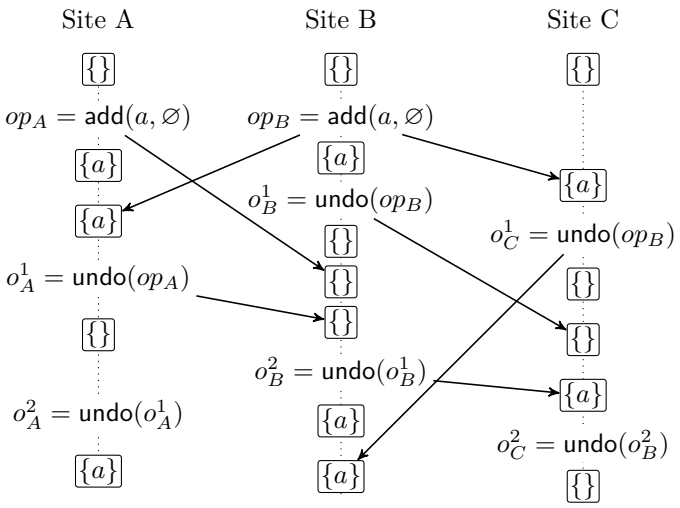


Figure 4 A scenario of concurrent undo operations.

5.2 Capturing concurrency and causality of undo operations

An application performs operations to modify its data. For example, the add operation adds an element into a set. We call these *normal operations*. In a distributed system, different sites may perform the same normal operations concurrently (or more specifically, the same *anonymous* operations described in §4.1). In Figure 4, site A and site B perform the same $\mathsf{add}(a)$ operation concurrently.

When the application undoes a normal operation, it cancels the effect of the modification. It can even further undo the undo (to achieve a redo), etc. We use op for a normal operation and o for any operation, which can be either a normal operation or an undo operation. When the application applies an undo operation on an earlier performed operation o, denoted as $o' = \mathsf{undo}(o)$, we say that o' is an *undo operation* that *directly undoes* o. An application can only directly undo an operation when it has observed the effect of that operation.

In Figure 4, o_A^1 directly undoes op_A, o_B^1 and o_C^1 directly undo op_B, o_A^2 directly undoes o_A^1, o_B^2 directly undoes o_B^1, and o_C^2 directly undoes o_B^2.

We relate the (normal or undo) operations with the same intention through the tie relation. An operation o_1 *ties* with operation o_2, denoted as $o_1 \sim o_2$, if one of the following holds: (i) $o_1 = o_2$, (ii) o_1 and o_2 are the same normal (anonymous) operations, (iii) $o_1 = \mathsf{undo}(o)$ and $o_2 = \mathsf{undo}(o)$, (iv) $o_1 = \mathsf{undo}(o_1')$, $o_2 = \mathsf{undo}(o_2')$ and $o_1' \sim o_2'$.

In Figure 4, $op_A \sim op_B$ because they are the same normal operations $\mathsf{add}(a)$, $o_B^1 \sim o_C^1$ because both directly undo the same operation op_B; $o_A^1 \sim o_B^1$ because $op_A \sim op_B$; $o_A^2 \sim o_B^2$ because $o_A^1 \sim o_B^1$.

▶ **Lemma 2** ($\sim$ properties). *The $\sim$ relation is reflexive, symmetric and transitive.*

Consequently, the tie relation partitions the operations into equivalence groups. For example, the equivalence groups in Figure 4 are $\{op_A, op_B\}$, $\{o_A^1, o_B^1, o_C^1\}$, $\{o_A^2, o_B^2\}$ and $\{o_C^2\}$.

One requirement on handling concurrent normal or undo operations is that the application should observe the same effect of tied operations.

The tie relation $\sim$ captures the concurrency of undo operations. The following undo-supersede relation captures the causality of undo operations. An operation o *undo-supersedes*

operation o', denoted as $o \succ_{\mathrm{u}} o'$, if one of the following holds: (i) $o = \mathsf{undo}(o')$, (ii) $o = \mathsf{undo}(o'')$ and $o'' \sim o'$, (iii) $o = \mathsf{undo}(o'')$ and $o'' \succ_{\mathrm{u}} o'$.

In Figure 4, $o_A^2 \succ_{\mathrm{u}} o_A^1$ because $o_A^2 = \mathsf{undo}(o_A^1)$; $o_A^2 \succ_{\mathrm{u}} o_B^1$ because $o_A^2 = \mathsf{undo}(o_A^1)$ and $o_A^1 \sim o_B^1$; $o_C^2 \succ_{\mathrm{u}} o_B^1$ because $o_C^2 = \mathsf{undo}(o_B^2)$ and $o_B^2 \succ_{\mathrm{u}} o_B^1$.

▶ **Lemma 3** ($\succ_{\mathrm{u}}$ properties). *The $\succ_u$ relation is irreflexive, asymmetric and transitive.*

For an operation o, its original operation*, denoted as $\mathsf{orig}(o)$, is a normal operation op, such that either (i) $o = op$, or (ii) $o = \mathsf{undo}(op)$, or (iii) $o = \mathsf{undo}(o')$ and $\mathsf{orig}(o') = op$.*

In Figure 4, $\mathsf{orig}(op_A) = \mathsf{orig}(o_A^1) = \mathsf{orig}(o_A^2) = op_A$, and $\mathsf{orig}(op_B) = \mathsf{orig}(o_B^1) = \mathsf{orig}(o_B^2) = \mathsf{orig}(\mathsf{o}_\mathsf{C}^1) = \mathsf{orig}(o_C^2) = op_B$.

Two operations o_1 and o_2 have the same origin*, denoted as $o_1 \overset{orig}{=} o_2$, if either $\mathsf{orig}(o_1) = \mathsf{orig}(o2)$ or $\mathsf{orig}(o_1) \sim \mathsf{orig}(o2)$.*

In Figure 4, $o_A^1 \overset{orig}{=} o_A^2$ because $\mathsf{orig}(o_A^1) = \mathsf{orig}(o_A^2)$; $o_A^1 \overset{orig}{=} o_C^2$ because $\mathsf{orig}(o_A^1) \sim \mathsf{orig}(o_C^2)$.

▶ **Lemma 4** (origin and undo relations). *Undo operations have the same origin iff they are related with tie or undo-supersede relations. Formally, $o_1 \overset{orig}{=} o_2 \Leftrightarrow o_1 \sim o_2 \vee o_1 \succ_u o_2 \vee o_2 \succ_u o_1$.*

For two concurrent undo operations o_1 and o_2 that have the same origin, a merge of o_1 and o_2, $\mathsf{merge}(o_1, o_2)$, should result in either (i) o_1 or o_2 if $o_1 \sim o_2$ (which one does not matter), (ii) o_1 if $o_1 \succ_u o_2$, or (iii) o_2 if $o_2 \succ_u o_1$.

When a site merges two concurrent operations, if the two operations tie with each other, they should have the same effect and the result of the merge can be either of them. In Figure 4, $o_A^1 \sim o_B^1$, hence $\mathsf{merge}(o_A^1, o_B^1) = o_A^1$ (or equally o_B^1). If one operation undo-supersedes the other, $o_1 \succ_u o_2$, o_1 has already seen the effect of o_2 and is causally dependent on o_2. Therefore the result of the merge is o_1. In Figure 4, $o_B^2 \succ_u o_C^1$ and o_B^2 has seen the effect of o_C^1 (which is equivalent to the effect of o_B^1 because $o_B^1 \sim o_C^1$), hence $\mathsf{merge}(o_C^1, o_B^2) = o_B^2$.

Now we define the undo length *of an operation o as:*

$$\mathsf{ulen}(o) \overset{def}{=} \begin{cases} 0 & \textit{if } o \textit{ is a normal operation} \\ \mathsf{ulen}(o') + 1 & \textit{if } o = \mathsf{undo}(o') \end{cases}$$

In Figure 4, $\mathsf{ulen}(op_A) = \mathsf{ulen}(op_B) = 0$, $\mathsf{ulen}(o_A^1) = \mathsf{ulen}(o_B^1) = \mathsf{ulen}(o_C^1) = 1$, $\mathsf{ulen}(o_A^2) = \mathsf{ulen}(o_B^2) = 2$, and $\mathsf{ulen}(o_C^2) = 3$.

▶ **Lemma 5** (undo length). *Let $o_1 \overset{orig}{=} o_2$. $\mathsf{ulen}(o_1) = \mathsf{ulen}(o_2)$ iff $o_1 \sim o_2$; $\mathsf{ulen}(o_1) > \mathsf{ulen}(o_2)$ iff $o_1 \succ_u o_2$.*

▶ **Lemma 6** (undo merge). *Let $o_1 \overset{orig}{=} o_2$ and $o = \mathsf{merge}(o_1, o_2)$. The undo length of o is $\mathsf{ulen}(o) = \mathsf{max}\big(\mathsf{ulen}(o_1), \mathsf{ulen}(o_2)\big)$.*

We could name the equivalence groups under $\sim$ in such a way that G_{op}^0 contains original normal operations and every operation in G_{op}^{n+1} directly undoes an operation in G_{op}^n. Then for any operation $o \in G_{op}^n$, $\mathsf{ulen}(o) = n$. For example, in Figure 4, $G_{\mathsf{add}(a)}^0 = \{op_A, op_B\}$, $G_{\mathsf{add}(a)}^1 = \{o_A^1, o_B^1, o_C^1\}$, $G_{\mathsf{add}(a)}^2 = \{o_A^2, o_B^2\}$ and $G_{\mathsf{add}(a)}^3 = \{o_C^2\}$.

In applications like editors, people often use the terms undo or redo with respect to the original normal operations. When $\mathsf{orig}(o) \sim op$, we say that

- *o undoes op* if either (i) $o = \mathsf{undo}(op)$, or (ii) $o = \mathsf{undo}(\mathsf{undo}(o'))$ and o' undoes op;
- *o redoes op* if $o = \mathsf{undo}(o')$ and o' undoes op.

In Figure 4, o_A^1, o_B^1, o_C^1 and o_C^2 undo $\mathsf{add}(a)$ (either op_A or op_B), whereas o_A^2 and o_B^2 redo $\mathsf{add}(a)$.

Obviously, if o undoes op, then $\mathsf{undo}(o)$ redoes op. Similarly, if o redoes op, then $\mathsf{undo}(o)$ undoes op.

▶ **Theorem 7** (undo-redo). *Given* $\mathsf{orig}(o) \sim op$,

- *o undoes op iff* $\mathsf{ulen}(o)$ *is a positive odd number;*
- *o redoes op iff* $\mathsf{ulen}(o)$ *is a positive even number.*

We can use the *undo-redo* theorem to answer the question in §5.1.

We omit the proofs of the lemmas and theorem in this section as they are trivial, simply by permutation on the different cases or by induction on undo lengths.

An application at a site always behaves according to the observation of its latest local state. An undo operation o is a *latest undo* of a normal operation op at a site, if $\mathsf{orig}(o) \sim op$ and there does not exist o' at the site such that $o' \succ_{\mathrm{u}} o$.

Locally, an application can only generate a normal operation, directly undo a normal operation if it has not been undone at the site, or directly undo a latest undo operation at the site. In Figure 4, when site B has received o_A^1, the latest undo operations of op_A (or equally op_B) are o_A^1 and o_B^1. Thus site B can only directly undo o_A^1 or o_B^1. It does not matter which of them to undo, because $o_A^1 \sim o_B^1$.

To incorporate the effect of a remote undo operation o, a site merges o with a latest operation o_l that has the same origin with o. If the remote operation o undo-supersedes the local operation o_l, the result of the merge is o and the site incorporates the effects of o; otherwise the result is o_l that the site has already incorporated.

6 Generically Supporting Undo for CRDTs

This section presents our second main contribution, our approach to generically supporting undo for existing CRDTs using the abstraction presented earlier in §5.

6.1 State Deltas as Operations

Every state in a state-based CRDT can be generated from a set of join-irreducible states (see §4.2). In Figures 1–3, states in boxes are join-irreducible. Given a mutator m, the states before and after applying m are s and $m(s)$. Let J_s and $J_{m(s)}$ be the sets of join-irreducible states that generate s and $m(s)$. The *state delta* caused by the execution of m on s is the set of join-irreducible states $J_{m(s)} - J_s$. For example, for the GSet CRDT (Figure 1), the state delta of the operation $\mathsf{add}(e, s)$ is $(s \cup \{e\}) - s = \{e\}$ when $e \notin s$. When e is already in s, the state delta is an empty set and no operation is actually executed.

It is a common and intuitive practice that a state-based CRDT is designed in such a way that every state delta consists of a single join-irreducible state. Or in the case of delta-state CRDTs, every delta-mutator returns a single join-irreducible state. For example, the state delta of $\mathsf{add}(e, s)$ of GSet is $\{e\}$ and the state delta of $\mathsf{inc}_i(s)$ of $\mathsf{GCounter}$ is $\{\langle i, s(i) + 1\rangle\}$. We observe that all delta-state CRDTs presented in [3] show this property. With such design, we can use join-irreducible states to represent operations of the CRDT.

In this paper, we assume that the state delta of a normal operation op consists of a single join-irreducible state, written as $\downarrow^{\delta} op$. Due to space limit, we do not deal with composite operations consisting of multiple join-irreducible states.

$$
\begin{aligned}
\mathsf{UState}(\mathcal{S}) &\stackrel{\text{def}}{=} S \hookrightarrow \mathbb{N} \\
\mathsf{reg}_u(s) &\stackrel{\text{def}}{=} \begin{cases} u\{s \mapsto u(s)+1\} & \text{if } s \in \mathsf{dom}(u) \\ u\{s \mapsto 0\} & \text{otherwise} \end{cases} \\
\mathsf{reg}^{\delta}_u(s) &\stackrel{\text{def}}{=} \begin{cases} \{\langle s, u(s)+1\rangle\} & \text{if } s \in \mathsf{dom}(u) \\ \{\langle s, 0\rangle\} & \text{otherwise} \end{cases} \\
u \sqcup u' &\stackrel{\text{def}}{=} \{\langle s, \mathsf{max}\big(u(s), u'(s)\big)\rangle \mid s \in \mathsf{dom}(u) \cup \mathsf{dom}(u')\} \\
\mathsf{undone}_u(s) &\stackrel{\text{def}}{=} \mathsf{odd}(u(s)) \\
\mathsf{valid}_u(s) &\stackrel{\text{def}}{=} s \in \mathsf{dom}(u) \wedge \mathsf{even}(u(s)) \\
\mathsf{valid}^{+}_{\mathsf{u}}(s) &\stackrel{\text{def}}{=} \bigwedge_{x \in \mathsf{dom}(u), x \sqsubseteq s} \mathsf{valid}_u(x)
\end{aligned}
$$

Figure 5 CRDT for undo states.

6.2 Undo-State CRDT

We maintain the undo states of operations as meta-data using the undo-state CRDT UState (Figure 5). For an existing CRDT with possible join-irreducible states $\mathcal{S}$, the undo state is a partial function $u\colon \mathcal{S} \hookrightarrow \mathbb{N}$. For a normal operation op of that CRDT, whose state delta is the join-irreducible state $s = \downarrow^{\delta} op$, $s \in \mathsf{dom}(u)$ means the operation op has been performed and $u(s)$ is the undo length of a latest undo operation of op (see §5.2 for the respective definitions). Notice that the bottom of $\mathbb{N}$, $\bot_{\mathbb{N}} = 0$. If an operation op has not been performed and thus $\downarrow^{\delta} op \notin \mathsf{dom}(u)$, applying $u(\downarrow^{\delta} op)$ (for example, when performing a join or a query), the result is 0 (§3).

For a normal operation op of the existing CRDT, $\downarrow^{\delta} op = s$, the operation $\mathsf{reg}_u(s)$ of UState registers the new latest undo state of op. The normal operation itself is registered with the addition of a new pair $\langle s, 0\rangle$ into the undo state. A new direct undo of a latest undo operation of op is registered with an incremental of $u(s)$ with one.

A join $\sqcup$ of two undo states u and u' merges the undo lengths of all operations that are registered in either u or u' (according to Lemma *undo merge* in §5.2).

Notice that UState is an anonymous CRDT. To see how this works, remember that we can partition the set of operations into equivalence groups under the tie relation $\sim$ (§5.2). Imagine that we register a new undo operation by adding it into the corresponding equivalence group. We can have an anonymous CRDT for the equivalence groups because they are grow-only sets. Using undo lengths in place of equivalence groups is just a way of compressing the undo states. The compression is possible because we are only interested in whether an equivalence group exists, rather than the specific elements in the groups. In addition, a site can only add an element in a new empty group, because it can only directly undo the latest undo operation of that site.

In Figure 4, after site C has incorporated received operation o^2_B, it sees the operations in equivalence groups $G^0_{\mathsf{add}(a)} = \{op_B\}$, $G^1_{\mathsf{add}(a)} = \{o^1_B, o^1_C\}$ and $G^2_{\mathsf{add}(a)} = \{o^2_B\}$. When performing o^2_C, it creates an empty group $G^3_{\mathsf{add}(a)}$ and adds o^2_C into it. Thereby $u(\{a\})$ in the UState becomes 3.

Another way to look at the undo state is to regard it as a log of the operations that have been performed. For every operation in the log, the recorded information is compressed into a single number, the undo length.

The predicate $\mathsf{undone}_u(s)$ states that the normal operation whose state delta is s is currently undone (according to Theorem *undo-redo* in §5.2).

The predicate $\mathsf{valid}_u(s)$ states that a state s in the existing CRDT is valid in u (i.e. $\mathsf{valid}_u(s)$ evaluates to True) if the corresponding normal operation has been performed (i.e. $s \in \mathsf{dom}(u)$), and the operation either has not been undone (i.e. $u(s) = 0$), or it has been undone but is finally redone (i.e. $u(s) > 0 \wedge \mathsf{even}(u(s))$.

The predicate valid_u^+ takes the dependencies of join-irreducible states into account. When a join-irreducible state becomes invalid due to undo (i.e. valid_u evaluates to False), all states depending on it also become invalid (i.e. valid_u^+ evaluates to False). For example, the state 3_A of $\mathsf{Gcounter}$ (Figure 2) depends on state 2_A. $\mathsf{valid}_u^+(3_A) = \mathsf{False}$ when $\mathsf{valid}_u(\{2_A\}) = \mathsf{False}$.

Notice that the states in GSet form an anti-chain. That is, every join-irreducible state is independent of any other join-irreducible state. For such CRDTs, valid_u^+ gives the same result as valid_u.

To compute the predicate valid_u^+, we need to find out the dependencies among join-irreducible states, using the links in the Hasse diagrams (i.e. the cover relation $\sqsubset_c$). For some CRDTs, the dependencies can be derived. For example, for $\mathsf{GCounter}$, $n_i \sqsubset_c (n+1)_i$ where $n \geq 0$. In case the dependencies cannot be derived, we have to materialize the dependencies, for instance, using a list or tree data structure.

6.3 Augmenting Existing CRDTs with Undo

For an existing CRDT T with possible states in $\mathcal{S}_T$, the CRDT augmented with undo support T_U is a composition of $\mathcal{S}_T$ and $\mathsf{UState}(\mathcal{J}(\mathcal{S}_T))$. Figure 6 shows the T_U CRDT.

The operation $\mathsf{do}_{\langle s,u\rangle}(op)$ performs a normal operation op in state s and registers op in undo state u. The operation $\mathsf{undo_latest}_{\langle s,u\rangle}(op)$ directly undoes the latest undo operation of op in state s: it registers the new latest undo in undo state u and has *no effect* on s. A site can only perform an undo when the original normal operation op has been performed or incorporated (i.e. the state delta $\Downarrow^{\delta} op$ is registered in u). Otherwise, performing an undo has no effect on undo state u.

To join two augmented states $\langle s, u\rangle$ and $\langle s', u'\rangle$, we join independently the states s and s' in $\mathcal{S}_T$ and the states u and u' in $\mathsf{UState}(\mathcal{J}(\mathcal{S}_T))$.

Queries in the original CRDT T are now performed on states transformed from augmented states. $\nu_u(s)$ defines a transformation that transforms a state s in the original CRDT using the undo state u. To see how it works, remember that the following holds for every state s in $\mathcal{S}_T$ (§4.2):

$$s = \bigsqcup \{x \in \mathcal{J}(\mathcal{S}_T) | x \sqsubseteq s\}$$

The transformation ν_u first filters out the invalid join-irreducible states and then joins the valid join-irreducible states to bring back the up-to-date state that reflects the undone effects.

The state transformation can be very costly if applied for every query. To address this, every site maintains a buffer of the transformed state. Every time the site updates the undo state, it also updates the buffered state. For example, when state $\{a\}$ of GSet becomes invalid, we remove element a from the buffered state. Indeed, the buffered states do not form a join-semilattice. This, however, does not lead to inconsistencies, because the buffered states are only local to the sites and are not propagated to remote sites.

$$
\begin{aligned}
\mathsf{T}_U &\stackrel{\text{def}}{=} \mathcal{S}_T \times \mathsf{UState}(\mathcal{J}(\mathcal{S}_T)) \\
\mathsf{do}_{\langle s,u\rangle}(op) &\stackrel{\text{def}}{=} \langle s \sqcup \downarrow^{\delta} op, \mathsf{reg}_u(\downarrow^{\delta} op)\rangle \\
\mathsf{do}^{\delta}_{\langle s,u\rangle}(op) &\stackrel{\text{def}}{=} \langle \downarrow^{\delta} op, \mathsf{reg}^{\delta}_u(\downarrow^{\delta} op)\rangle \\
\mathsf{undo_latest}_{\langle s,u\rangle}(op) &\stackrel{\text{def}}{=} \begin{cases} \langle s, \mathsf{reg}_u(\downarrow^{\delta} op)\rangle & \text{if } \downarrow^{\delta} op \in \mathsf{dom}(u) \\ \langle s, u\rangle & \text{otherwise} \end{cases} \\
\mathsf{undo_latest}^{\delta}_{\langle s,u\rangle}(op) &\stackrel{\text{def}}{=} \begin{cases} \langle \bot, \mathsf{reg}^{\delta}_u(\downarrow^{\delta} op)\rangle & \text{if } \downarrow^{\delta} op \in \mathsf{dom}(u) \\ \langle \bot, \bot\rangle & \text{otherwise} \end{cases} \\
\langle s, u\rangle \sqcup \langle s', u'\rangle &\stackrel{\text{def}}{=} \langle s \sqcup s', u \sqcup u'\rangle \\
\nu_u(s) &\stackrel{\text{def}}{=} \bigsqcup \{x \in \mathcal{J}(\mathcal{S}_T) | x \sqsubseteq s \wedge \mathsf{valid}^{+}_u(s)\} \\
\mathsf{query}_u(\dots, s, \dots) &\stackrel{\text{def}}{=} \mathsf{query}(\dots, \nu_u(s), \dots)
\end{aligned}
$$

Figure 6 CRDT augmented with undo.

Now we use some examples to illustrate how the augmentation works.

We first augment GSet (Figure 1) to GSet_U for undo support. $\mathsf{add}_u(e, s)$ performs $\mathsf{do}_{\langle \mathsf{s,u}\rangle}(\mathsf{add(e)})$. The query in_u in GSet_U is equivalent to the following:

$$\mathsf{in}_u(e, s) \stackrel{\text{def}}{=} e \in s \wedge \neg \mathsf{undone}_u(\{e\})$$

The query now takes the undo effect into account. In Figure 4, the latest undo operation of $\mathsf{add}(a)$ at site B is o^2_B. Because $\mathsf{ulen}(o^2_B) = 2$, the undo state at site B is $\{\langle\{a\}, 2\rangle\}$. Therefore the join-irreducible state $\{a\}$ is valid and $\mathsf{in}_u(a, \{a\})$ evaluates to True. On the other hand, the latest undo operation of $\mathsf{add}(a)$ at site C is o^2_C. Because $\mathsf{ulen}(o^2_C) = 3$, the undo state at site C is $\{\langle\{a\}, 3\rangle\}$. Therefore the join-irreducible state $\{a\}$ is invalid and the query $\mathsf{in}_u(a, \{a\})$ evaluates to False.

Now, let us augment $\mathsf{2P_{\mathbb{B}}Set}$ (Figure 3) with undo support. In the first scenario, a site adds a, b, removes a and then undoes the removal. The state in $\mathsf{2P_{\mathbb{B}}Set}$ is $s^1 = \{\langle a, \mathsf{True}\rangle, \langle b, \mathsf{False}\rangle\}$ and the undo state is $u^1 = \{\langle\{\langle a, \mathsf{False}\rangle\}, 0\rangle, \langle\{\langle a, \mathsf{True}\rangle\}, 1\rangle, \langle\{\langle b, \mathsf{False}\rangle\}, 0\rangle\}$. The predicate $\mathsf{valid}_{u^1}(\{\langle a, \mathsf{True}\rangle\}) = \mathsf{False}$. Transforming the state results in $s^1_u = \nu_{u^1}(s^1) = \{\langle a, \mathsf{False}\rangle\} \cup \{\langle b, \mathsf{False}\rangle\} = \{\langle a, \mathsf{False}\rangle, \langle b, \mathsf{False}\rangle\}$. The results of queries on s^1_u are as expected, $\mathsf{in}(a, s^1_u) = \mathsf{in}(b, s^1_u) = \mathsf{True}$. That is, both a and b are in the set.

In the second scenario, a site adds a, b, removes a and then undoes the addition of a. The state in $\mathsf{2P_{\mathbb{B}}Set}$ is $s^2 = \{\langle a, \mathsf{True}\rangle, \langle b, \mathsf{False}\rangle\}$ and the undo state is $u^2 = \{\langle\{\langle a, \mathsf{False}\rangle\}, 1\rangle, \langle\{\langle a, \mathsf{True}\rangle\}, 0\rangle, \langle\{\langle b, \mathsf{False}\rangle\}, 0\rangle\}$. Observe that $s^2 = s^1$ and $u^2 \neq u^1$, meaning that an undo does not alter the state of the original CRDT. The predicates $\mathsf{valid}_{u^2}(\{\langle a, \mathsf{False}\rangle\}) = \mathsf{False}$ and $\mathsf{valid}^{+}_{u^2}(\{\langle a, \mathsf{True}\rangle\}) = \mathsf{False}$. Transforming the state results in $s^2_u = \nu_{u^2}(s^2) = \{\langle b, \mathsf{False}\rangle\}$. Again, the results of queries on s^2_u are as expected, $\mathsf{in}(a, s^2_u) = \mathsf{False}$ and $\mathsf{in}(b, s^1_u) = \mathsf{True}$. That is, b is in the set but a is not (as if a had never been added).

The last examples with $\mathsf{2P_{\mathbb{B}}Set}$ indicate that the generic undo support works well with CRDTs that themselves support inverse operations.

$$
\begin{aligned}
\mathsf{Doc}(C) &\stackrel{\text{def}}{=} C \hookrightarrow \mathcal{P}(\mathbb{I}) \\
\mathsf{ins}(c, m) &\stackrel{\text{def}}{=} m\{c \mapsto \varnothing\} \\
\mathsf{ins}^{\delta}(c, m) &\stackrel{\text{def}}{=} \{\langle c, \varnothing\rangle\} \\
\mathsf{del}_i(c, m) &\stackrel{\text{def}}{=} \begin{cases} m\{c \mapsto m(c) \cup \{i\})\} & \text{if } c \in \mathsf{dom}(m) \\ m & \text{otherwise} \end{cases} \\
\mathsf{del}_i^{\delta}(c, m) &\stackrel{\text{def}}{=} \begin{cases} \{\langle c, \{i\}\rangle\} & \text{if } c \in \mathsf{dom}(m) \\ \bot & \text{otherwise} \end{cases} \\
m \sqcup m' &\stackrel{\text{def}}{=} \{\langle c, m(c) \cup m'(c)\rangle \mid c \in \mathsf{dom}(m) \cup \mathsf{dom}(m')\} \\
\mathsf{visible}(c, m) &\stackrel{\text{def}}{=} c \in \mathsf{dom}(m) \wedge m(c) = \varnothing
\end{aligned}
$$

Figure 7 CRDT for a collaborative text editor.

7 Collaborative Editing with Undo Support

In this section, we show a practical application of the undo support for collaborative editing.

The collaborative editing system is based on the CRDT reported in [29]. It consists of several peers, each of which has a replica of the shared document under editing. At each peer, a user edits the local copy of the document via the *document view*, which is simply a string of characters. Under the hood, there is a *document model*, which is a CRDT of characters. The view is the concatenation of *visible* characters in the model. We could regard the view as the buffer of transformed state discussed in §6.3.

Figure 7 shows the (simplified) CRDT of the document model. The CRDT is a function from the set of characters C to the power set of site identifiers $\mathbb{I}$.

The characters of the CRDT have globally unique and ordered identifiers that are specific to the sites that inserted the character ([26, 4]). Therefore the Doc CRDT is named – every character is unique and cannot be concurrently inserted at different peers. However, different peers can concurrently delete the same character.

When a character c is inserted, c maps to an empty set. When site i deletes c, i is added to the set that c maps to. A character c is visible in the document if it is inserted but not deleted, that is, when c is in the domain and maps to the empty set.

To support undo, we simply augment Doc to Doc_U. The designer of the Doc CRDT does not need to manually design anything in addition. In the augmented CRDT, $\mathsf{visible}_u$ is equivalent to the following:

$$
\mathsf{visible}_u(c, m) \stackrel{\text{def}}{=} \mathsf{valid}_u(\{\langle c, \varnothing\rangle\}) \wedge \bigwedge_{i \in m(c)} \neg\mathsf{valid}_u(\{\langle c, \{i\}\rangle\})
$$

A character c is visible in the document if it is inserted and the insertion is not undone, and if it is deleted, all deletions are undone.

To see why a character should be visible only when all deletions are undone, consider the situation where site A deletes a character "x" and then undoes the deletion. Meanwhile, site B also deletes "x". The final effect should be as if site A had done nothing and site B performed a deletion. So character "x" should not appear in the document.

Some researchers (for example [22]) regard concurrent deletions of the same character as the same operation (which may lead to some confusing semantics of the undo of string-wise operations [29]). We could achieve this by using partial function $C \hookrightarrow \mathbb{B}$ (similar to the $2\mathsf{P}_{\mathbb{B}}\mathsf{Set}$ CRDT in Figure 3) rather than $C \hookrightarrow \mathcal{P}(\mathbb{I})$. With this re-design, a character is visible when only one deletion is undone (because all deletions of the same character are regarded as the same anonymous operation).

8 Related Work

Supporting inverse operations was already a topic when CRDTs were first presented [23], such as a counter that can be both incremented and decremented, a set where elements can be both added and removed, etc. The CRDT designer has to design new customized CRDTs in order to support inverse operations. A common problem is that the designer has to decide a "winner" between an operation and its inverse counterpart, for example, a removal always wins (see §4.3 for an example). Furthermore, a "loser" has never got a chance to "win back".

A causal CRDT [3] associates causal contexts with every operation (or element) to achieve the effect such as adding a removed element back to a set. The CRDT designer has to write a new causal CRDT for a given CRDT to get this support. Furthermore, maintaining causal contexts for every operation could be costly when the number of replicas is large.

Our work provides a generic support of undo for any (to our knowledge) state-based CRDT. The CRDT designer does not need to write a new specialize CRDT to get the undo feature. Furthermore, the undo state for an operation is only a single number.

Undo has been a research topic in the area of collaborative editing for decades ([10, 21, 22, 24, 25, 27]). Most of the work was not able to define the semantics of undo and redo operations with respect to concurrency and causality, and therefore showed incorrect behavior as discussed in §5.1.

The abstraction we proposed, although seemingly simple, correctly captures the semantics of concurrent undo an redo operations and does not have the aforementioned issues.

The Doc CRDT (§7) is a simplification of the work presented in [29]. The model CRDT in [29] represents undo relations using equivalence classes (§5.2) rather than the more compact undo lengths. This allows the editor to support additional features such as displaying who performed a particular undo operation.

The system presented in [9] supports cascading undo of selected operations by explicitly defining dependencies among operations using a process specification language. In our work, operation dependencies are implied by the state order $\sqsubset$ of join-semilattices.

9 Conclusion

In this work, we have presented how to provide undo features to existing CRDTs. Our work consists of two major parts.

The first part is an abstraction that captures the semantics concurrent undo and redo operations using equivalence classes. The abstraction can be compacted into single numbers (undo lengths) that are straightforward to implement in practice. The abstraction is generally applicable (not restricted to CRDTs) to any system that demands concurrent undo and redo of earlier performed operations.

The second part is a generic approach to augmenting existing state-based CRDTs with the capability of undo. The augmentation transforms the states in an original CRDT to the ones with the undo effects. Unmodified queries can be applied to the transformed states.

The states of the augmented CRDTs converge eventually, because the state transformation is local to the replicas and does not propagate to the global system.

We have shown a practical application of our work in collaborative editing.

Operation-based CRDTs have also found their ways in applications that demand undo support. Supporting undo features for operation-based CRDTs is an open research topic.

References

1. Gregory D Abowd and Alan J Dix. Giving undo attention. *Interacting with Computers*, 4(3):317–342, 1992.
2. Deepthi Devaki Akkoorath, Alejandro Z. Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno M. Preguiça, and Marc Shapiro. Cure: Strong Semantics Meets High Availability and Low Latency. In *36th IEEE International Conference on Distributed Computing Systems, (ICDCS)*, pages 405–414, 2016.
3. Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Delta state replicated data types. *J. Parallel Distrib. Comput.*, 111:162–173, 2018.
4. Luc André, Stéphane Martin, Gérald Oster, and Claudia-Lavinia Ignat. Supporting adaptable granularity of changes for massive-scale collaborative editing. In *CollaborateCom*, 2013.
5. Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. Making Operation-Based CRDTs Operation-Based. In *14th IFIP WG 6.1 International Conference Distributed Applications and Interoperable Systems (DAIS)*, volume 8460 of *Lecture Notes in Computer Science*, pages 126–140. Springer, 2014.
6. Annette Bieniusa, Marek Zawirski, Nuno M. Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. A optimized conflict-free replicated set. *Rapport de recherche*, 8083, October 2012.
7. Eric A. Brewer. CAP twelve years later: How the "rules" have changed. *IEEE Computer*, 45(2):23–29, 2012.
8. Russell Brown, Sean Cribbs, Christopher Meiklejohn, and Sam Elliott. Riak DT map: a composable, convergent replicated dictionary. In *The First Workshop on the Principles and Practice of Eventual Consistency*, pages 1–1, 2014.
9. Aaron G. Cass and Chris S. T. Fernandes. Using Task Models for Cascading Selective Undo. In Karin Coninx, Kris Luyten, and Kevin A. Schneider, editors, *Task Models and Diagrams for Users Interface Design*, pages 186–201. Springer Berlin Heidelberg, 2007.
10. Jean Ferrié, Nicolas Vidot, and Michèle Cart. Concurrent Undo Operations in Collaborative Environments Using Operational Transformation. In *CoopIS/DOA/ODBASE (1)*, volume 3290 of *Lecture Notes in Computer Science*, pages 155–173. Springer, 2004.
11. Armando Fox and Eric A. Brewer. Harvest, Yield and Scalable Tolerant Systems. In *The Seventh Workshop on Hot Topics in Operating Systems*, pages 174–178, 1999.
12. Hector Garcia-Molina and Kenneth Salem. Sagas. In *1987 ACM SIGMOD International Conference on Management of Data*, pages 249–259. ACM, 1987.
13. Vijay K. Garg. *Introduction to Lattice Theory with Computer Science Applications*. Wiley, 2015.
14. Seth Gilbert and Nancy Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News*, 33(2):51–59, 2002.
15. Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno M. Preguiça, and Rodrigo Rodrigues. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 265–278, 2012.
16. Christopher Meiklejohn and Peter Van Roy. Lasp: a language for distributed, coordination-free programming. In *the 17th International Symposium on Principles and Practice of Declarative Programming*, pages 184–195, 2015.

17 Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Data consistency for P2P collaborative editing. In *CSCW*, pages 259–268. ACM, 2006. doi:10.1145/1180875.1180916.

18 Douglas Stott Parker Jr., Gerald J. Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David A. Edwards, Stephen Kiser, and Charles S. Kline. Detection of Mutual Inconsistency in Distributed Systems. *IEEE Trans. Software Eng.*, 9(3):240–247, 1983.

19 Nuno M. Preguiça, Marek Zawirski, Annette Bieniusa, Sérgio Duarte, Valter Balegas, Carlos Baquero, and Marc Shapiro. SwiftCloud: Fault-Tolerant Geo-Replication Integrated all the Way to the Client Machine. In *33rd IEEE International Symposium on Reliable Distributed Systems Workshops (SRDS Workshops 2014)*, pages 30–33, 2014.

20 Nuno M. Preguiça, Joan Manuel Marquès, Marc Shapiro, and Mihai Letia. A Commutative Replicated Data Type for Cooperative Editing. In *29th IEEE International Conference on Distributed Computing Systems, (ICDCS)*, pages 395–403, 2009.

21 Matthias Ressel and Rul Gunzenhäuser. Reducing the problems of group undo. In *GROUP*, pages 131–139. ACM, 1999.

22 Bin Shao, Du Li, and Ning Gu. An algorithm for selective undo of any operation in collaborative applications. In *GROUP*, pages 131–140. ACM, 2010.

23 Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-Free Replicated Data Types. In *13th International Symposium on Stabilization, Safety, and Security of Distributed Systems, (SSS 2011)*, pages 386–400, 2011.

24 Chengzheng Sun. Undo as concurrent inverse in group editors. *ACM Trans. Comput.-Hum. Interact.*, 9(4):309–361, 2002.

25 David Sun and Chengzheng Sun. Context-Based Operational Transformation in Distributed Collaborative Editing Systems. *IEEE Trans. Parallel Distrib. Syst.*, 20(10):1454–1470, 2009. doi:10.1109/TPDS.2008.240.

26 Stéphane Weiss, Pascal Urso, and Pascal Molli. Logoot: A Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks. In *29th IEEE International Conference on Distributed Computing Systems, (ICDCS)*, pages 404–412, 2009.

27 Stéphane Weiss, Pascal Urso, and Pascal Molli. Logoot-Undo: Distributed Collaborative Editing System on P2P Networks. *IEEE Trans. Parallel Distrib. Syst.*, 21(8):1162–1174, 2010. doi:10.1109/TPDS.2009.173.

28 Weihai Yu. Supporting string-wise operations and selective undo for peer-to-peer group editing. In *GROUP*, pages 226–237. ACM, 2014.

29 Weihai Yu, Luc André, and Claudia-Lavinia Ignat. A CRDT Supporting Selective Undo for Collaborative Text Editing. In *15th IFIP WG 6.1 International Conference Distributed Applications and Interoperable Systems (DAIS)*, volume 9038 of *Lecture Notes in Computer Science*, pages 193–206. Springer, 2015.

In Search of the Fastest Concurrent Union-Find Algorithm

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

Alexander Fedorov
Higher School of Economics, St. Petersburg, Russia
JetBrains, St. Petersburg, Russia
afedorov2602@gmail.com

Nikita Koval
IST Austria, Klosterneuburg, Austria
JetBrains, St. Petersburg, Russia
ndkoval@ya.ru

Abstract

Union-Find (or Disjoint-Set Union) is one of the fundamental problems in computer science; it has been well-studied from both theoretical and practical perspectives in the sequential case. Recently, there has been mounting interest in analyzing this problem in the concurrent scenario, and several asymptotically-efficient algorithms have been proposed. Yet, to date, there is very little known about the practical performance of concurrent Union-Find.

This work addresses this gap. We evaluate and analyze the performance of several concurrent Union-Find algorithms and optimization strategies across a wide range of platforms (Intel, AMD, and ARM) and workloads (social, random, and road networks, as well as integrations into more complex algorithms). We first observe that, due to the limited computational cost, the number of induced cache misses is the critical determining factor for the performance of existing algorithms. We introduce new techniques to reduce this cost by storing node priorities implicitly and by using plain reads and writes in a way that does not affect the correctness of the algorithms. Finally, we show that Union-Find implementations are an interesting application for Transactional Memory (TM): one of the fastest algorithm variants we discovered is a sequential one that uses coarse-grained locking with the lock elision optimization to reduce synchronization cost and increase scalability.

2012 ACM Subject Classification Theory of computation → Concurrent algorithms; Computing methodologies → Concurrent algorithms; Theory of computation → Graph algorithms analysis

Keywords and phrases union-find, concurrency, evaluation, benchmarks, hardware transactional memory

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.15

Related Version https://arxiv.org/abs/1911.06347

1 Introduction

Union-Find – the problem of maintaining the global connectivity structure of a set of integer elements based on their pair-wise connectivity – is a fundamental problem in computer science. This problem is alternatively known as Disjoint-Set Union (DSU), and is a key component of several classic algorithms, such as Kruskal's [12] and Boruvka's [6] algorithms for finding minimum spanning trees (MSTs), maintaining connected components and finding loops under edge additions, or finding strongly-connected components in directed graphs.

In its classic formulation, the sequential DSU problem assumes a finite ground set S, upon whose elements we perform the following operations:

- `SameSet(u, v)` checks whether two elements `u` and `v` are in the same set;
- `Union(u, v)` merges the sets to which `u` and `v` are currently assigned;

© Dan Alistarh, Alexander Fedorov, and Nikita Koval;
licensed under Creative Commons License CC-BY
23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller; Article No. 15; pp. 15:1–15:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

- `Find(u)` returns the representative of the set in which `u` is located. This representative must be the same for all elements from the corresponding subset.

It is worth noting that in sequential case `SameSet` can be easily implemented via two `Find` invocations, on `u` and `v`, and checking whether the representatives coincide. Further optimizations exist to implement this method more efficiently; we discuss them in Section 4.2.

This classic problem is known to have a rich theoretical structure. A masterclass by Tarjan linked the upper bound on the worst-case time complexity of the problem to the inverse Ackermann function [18], followed by a matching lower bound for a restricted case [19], which was later extended to arbitrary algorithms to show optimality [13]. Later, Tarjan and Van Leeuwen [17] performed one of the first worst-case analyses for compaction heuristics in the context of Disjoint-Set Union (DSU), exposing the highly non-trivial fact that, with appropriate path compaction and linking heuristics, the problem can be solved in $\mathcal{O}(m \cdot \alpha(n, \frac{m}{n}))$ time complexity, where n is the number of elements, m is the number of operations, and α is a functional inverse of Ackermann's function.

Patwary, Blair, and Manne [15] were the first to perform an in-depth experimental study of sequential strategies. Their work clearly showed the importance of minimizing the number of reads from memory on the performance of DSU, since the various strategies have very limited computational demands. Their experiments exposed the fact that the fastest sequential algorithm was the one designed by Rem in 1976 [8], which we describe in detail in the following sections.

The first concurrent algorithm for DSU was proposed by Cybenko, Allen, and Polito [7], whose key idea was using a spin-lock for write operations. Years later, Anderson and Woll proposed a *wait-free* concurrent algorithm [5], which is roughly a concurrent generalization of one of the linking-and-compression strategies studied by Tarjan and Van Leeuwen [17]. Their paper claims a worst-case upper bound for the algorithm of $\Theta(m \cdot (\alpha(m, 0) + p))$, where p is the number of parallel processes, and m is the number of operations; it was later observed that this proof is correct only under the non-standard assumption that threads cannot be preempted in between some certain operations in the algorithm [10].

Recently, Jayanti and Tarjan presented a set of correct and asymptotically-efficient concurrent DSU algorithms, which use fixed random priorities rather than ranks, based on the randomized sequential algorithm of [9]. These algorithms achieve a (total) work complexity upper bound of $\mathcal{O}(m \cdot (\alpha(n, \frac{m}{p}) + \log(\frac{np}{m} + 1)))$. Recent work by these authors, in collaboration with Boix-Adserà, suggested an algorithm with the same total work complexity bounds, but showed that this is optimal for a class of natural "symmetric algorithms" [11].

Our Contribution

Motivated by the significant recent interest in the concurrent DSU problem, as well as by its numerous practical applications, in this paper we perform the first thorough study of the practical performance of concurrent DSU implementations. While our focus is to implement and study existing algorithmic proposals, along the way we discover new optimizations and algorithmic insights. We start from the basic observation that, given the simple structure of most algorithms for DSU, memory access and synchronization costs will be the dominating factors behind practical performance. With this in mind, we analyze the performance of several classic baselines, and propose a host of optimizations to specifically reduce the impact of these factors.

We perform a range of experiments across several architectures, algorithm variants and optimizations, synchronization primitives, as well as workloads, to determine the fastest concurrent DSU algorithm. We provide a wide range of results and discuss our findings across several dimensions in detail in Section 5. In brief, the variant which appears to be "fastest" for most of the settings considered is an optimized sequential algorithm variant that

```
1  func ParallelMST(G: Graph) {
2    mst = {} // empty set initially
3    while mst.Size < THRESHOLD(G.Size) {
4      shortestEdges: Edge[G.Size] // shortest edges for each component
5      // Phase 1. Find the shortest edges for each vertex
6      parallel for e in G.Edges {
7        if SameSet(e.from, e.to) { G.Edges.remove(e); continue }
8        UpdateIfShorter(&shortestEdges[Find(e.from)], e)
9        UpdateIfShorter(&shortestEdges[Find(e.to)], e)
10     }
11     // Phase 2. Unite components and update MST
12     parallel for u in G.Vertices {
13       Unite(shortestEdges[u].from, shortestEdges[u].to)
14       mst += shortestEdges[u]
15     }
16   }
17   return mst + SequentialMST(graph)
18 }
```

Listing 1 Parallel Boruvka's algorithm pseudo-code used in our experiments.

leverages HTM (lock elision) for high path compaction while minimizing synchronization cost.

2 Experimental Setup

For our evaluation, we use two classic graph algorithms based on DSU data structures. The first maintains the *connected components* for a given graph. In this case, for benchmarking, we randomly split graph edges between threads, and also mark whether the `Union` of the `SameSet` operation should be executed with a given edge as a parameter. The set of graphs we consider is presented in Table 1.

The second benchmark is a parallel version of the *Boruvka's algorithm* [6] for finding the minimum spanning tree. (The pseudo-code of the algorithm we use in the experiments is presented in Listing 1.) This algorithm performs at most $\log n$ iterations (where n is number of vertices), each is split into two phases. During the first phase, the algorithm finds the shortest adjacent edge for each vertex, removing the ones that connect vertices from the same component, the set of which is maintained by the DSU data structure. Here, the `UpdateIfShorter` function atomically checks whether the already stored edge is longer than the specified one, and replaces it in this affirmative case. During the second phase, the algorithm goes through all the representatives of the components and adds the corresponding shortest edges to the MST. The main idea of the concurrent algorithm is that both phases can be performed in parallel, with synchronization between them. However, if the number of remaining components is small, it is better to complete the work sequentially. Thus, we perform the parallel part until the already built MST part exceeds some size threshold. We consider only the parallel part in the benchmark.

Graph Inputs. We evaluate the algorithms above on a range of real-world and synthetic graphs. The connected components maintenance algorithm has linear time complexity (assuming for simplicity that the DSU operations work in constant time), while Boruvka's algorithm requires logarithmic time to complete, which is why we test it on relatively smaller graphs. Nevertheless, both benchmarks use road and social network graphs, as well as random ones with similar properties. The list of graphs we use for the connected components and Boruvka's algorithm benchmarks is presented in Tables 1 and 2 respectively.

The first graphs in both tables represent the USA road network, Central and West parts.

Table 1 Graphs for the connected components benchmark.

Graph	Vertices(M)	Edges(M)	Description
USA-ROADS	14.1	16.9	A graph with roads of the central part of the USA
LIVE-JOURNAL	4	34.7	A graph of ground-truth communities in LiveJournal social network
POKEC	1.6	30.5	A graph of friendship relations in Pokec social network
RANDOM	2.5	30	A random graph generated with the Erdős-Rényi model.
HIGH-CONTENTION	12	26.8	A synthetic graph for detecting "bad" DSU versions

Table 2 Graphs for the Boruvka's algorithm benchmark.

Graph	Vertices(M)	Edges(M)	Description
USA-ROADS	6.2	7.6	A graph with roads of the western part of the USA
BERKELEY-STANFORD	6.9	7.6	Hyperlinks between the Berkeley and the Stanford domains
INTERNET-TOPOLOGY	1.7	11.1	Internet topology graph, from traceroutes run daily in 2005
RANDOM	1	10	A random graph generated with the Erdős-Rényi model
HIGH-CONTENTION	4	8.9	A synthetic graph for detecting "bad" DSU versions

The next two graphs for the connected components benchmark are social network graphs, while the ones for the Boruvka's benchmark are Internet topology graphs. Also, we use two large synthetic graphs. The first one is a randomly-generated Erdoes-Renyi graph, with a specified number of vertices and edges ($2.5M$ vertices and $30M$ edges for the connected components, and $1M$ vertices and $10M$ edges for the Boruvka's algorithm); it is denoted as `RANDOM` in our experiments. The second synthetic graph is folklore for detecting "bad" DSU implementations in programming contests – most of the edges are incident to a small number of nodes; it is denoted as `HIGH-CONTENTION`.

Hardware. We perform the experiments on Intel, AMD, and ARM platforms; all the machines we used have several sockets, which induces Non-Uniform Memory Access (NUMA) effects. The detailed specifications are as follows:

- **Intel**. We used our local machine with 4 sockets, Intel Xeon Gold 6150 with 18 cores per socket, and hyperthreading enabled, for 144 hardware threads in total.
- **AMD**. We used a general-purpose Amazon AWS [1] instance with 6 sockets, 8-core AMD EPYC 7571 processors with hyperthreading enabled in each; 96 cores total.
- **ARM**. We used an instance on the Packet [3] cloud service with 2 sockets of 48-core ARM Cavium ThunderX processors; 96 cores total. The ARM memory model is more relaxed than the TSO one on Intel and AMD architectures, which would lead us to expect higher scalability.

Software. All algorithms and benchmarks are implemented either in Java or Kotlin, and compiled to the JVM byte-code; we use OpenJDK 11.0.4 with Ubuntu OS on all platforms. To avoid problems related to JIT compilers and reproducibility/benchmarking, we use the Java Microbenchmark Harness (JMH) library to run our benchmarks and collect statistics [2].

3 Sequential Implementations via Compressed Trees

We begin with an overview of sequential implementations, which are loosely based on the idea of maintaining a *compressed forest of trees* [18]. Each tree in this data structure corresponds to the membership of one set, where the root of the tree acts as its representative. The trees are implemented by maintaining parents for each element; thus, the data structure stores an array of parent links. Roots of the trees have their parent links point to themselves. Thus, in order to implement the `Find` operation, the algorithm "climbs" using parent links until it reaches the root. At the same time, the `Union` operation takes the corresponding tree roots as representatives, checks whether they coincide (finishing the operation in that case), and unites the sets by pointing one root to another. While the algorithm seems straightforward, it requires some heuristics to guarantee good time complexity.

Linking Strategies. When the `Union` operation decides to merge two different sets, it either changes the parent link of the first element to the second one or vice versa. Intuitively, we want to maintain the tree height as small as possible, so that `Find` operation works efficiently. The standard procedure is defining priorities on roots, so that the root with the lower priority is "hung" under the root with higher priority. The following definitions of priority are usually employed:

- *Tree size.* Each root maintains the size of its tree, and smaller roots are pointed to larger ones. This serves as a way of balancing the tree.
- *Rank.* Since we aim at optimizing the height, it is reasonable to store the height as a priority. When the ranks of the trees to be united are different, then the smaller tree is pointed to the larger one, and the ranks remain the same. However, when the ranks coincide, an arbitrary one is chosen as a new root, and its rank is incremented.
- *Random.* Another method to choose a new root is using a set of fixed random initial priorities; Tarjan et. al. first analyzed this technique in the sequential case [9], and after it Jayanti and Tarjan made an analysis for the concurrent case [10].

Path Compaction. Another way to make the data structure faster is making the tree "flatter" by shortening paths between nodes and the roots. In particular, notice that during the `Find` operation, the parent links can be changed to the higher ones without breaking correctness; this way, the algorithm reaches the root faster on the next `Find` invocation. Here are several strategies which are usually employed to compact trees:

- *Compression.* Once the root is found, all the elements on the search path can update their parent pointers to the root. The simplest implementation uses recursion and performs these updates in the last-in-first-out order. The disadvantage of this technique is that it requires performing two passes, from u to the root and back to u; thus, in practice, this strategy produces extra cache misses.
- *Splitting.* This technique updates the parent links during the only traversal to the root. On each step, it reads both the parent and the grandparent of the current element, and updates its parent link to the grandparent. Thus, it compresses the paths from all visited elements to the root by one.
- *Halving.* This technique is similar to splitting, but after each step, each node is linked to the grandparent. Thus, parents of only half of the elements on the path are updated.

These compaction optimizations can guarantee $\mathcal{O}(\log n)$ amortized time complexity of the `Find` operation, while the linking techniques guarantee $\mathcal{O}(\log n)$ worst-case time complexity (probabilistic in case of random priorities). However, using both techniques at the same time, linking by priority and path compaction, guarantees $\mathcal{O}(m \cdot \alpha(n, \frac{m}{n}))$ time complexity on average, where n is the number of elements, m is the number of operations, and α is a functional inverse of Ackermann's function.

```
1 func Union(u, v) {
2   u = Find(u); v = Find(v)
3   if u == v: return
4   if rank[u] <= rank[v]:
5     parent[u] = v
6   else if rank[u] > rank[v]:
7     parent[v] = u
8   if rank[u] == rank[v]: rank[v]++
9 }
10 func Find(u) = while (true) {
11   p = parent[u]
12   gp = parent[p]
13   if p == gp: return p
14   parent[u] = gp
15   u = gp
16 }
```

Listing 2 Sequential DSU with rank-based priorities using path compaction via halving.

Implementation Example. Listing 2 provides a pseudo-code of the DSU algorithm with ranks as priorities and path compaction via halving. Here we assume that the `SameSet` function is implemented with two `Find` invocations.

Evaluation. We evaluated all the combinations of linking and path compaction strategies. Similarly to the work by Patwary et al., we store priorities only for roots [15]. Thus, it is possible to use a single register to store either parent or priority – we only have to reserve one bit as a mark whether the element is a root or not; we show in the extended version of this paper [4] that this optimization significantly improves the performance.

In addition to the standard linking strategies, we suggest using a pseudo-random one, which essentially shuffles priorities in the range $1..n$ among n elements – the following well-known hashing formula due to Knuth is used:

$$priority(x) = (x + \texttt{SHIFT}) \times \texttt{BIG_PRIME} \mod N.$$

It is worth noting that `SHIFT` can be chosen randomly to increase the algorithm's robustness against some adversary to make the algorithm work slow.

Figure 1 shows the results of all the 12 combinations on both connected components and Boruvka's algorithm benchmarks; we used our local Intel machine for this experiment.

Rank priorities are approximations for tree sizes. We predictably do not see significant difference between these linking strategies; however, using ranks requires updates only if two trees in the `Unite` operation have the same ranks. Thus, we do not consider the size-based strategy in further concurrent tests – we believe that it cannot outperform the rank one due to more memory updates.

The suggested pseudo-random priorities slightly outperform the real random ones in almost all scenarios (time of random generation was not considered). We believe that this is a consequence of the fact that we do not need to store pseudo-random priorities at all and that the pseudo-random method shuffles elements as evenly as possible. Thus, we use only pseudo-random priorities in further experiments. As for the comparison between pseudo-random and rank-based priorities, we do not see one clearly and consistently outperform the other in many scenarios. Therefore, we consider it important to test both these strategies in a concurrent environment. Similarly, we keep all linking strategies for further experimentation.

4 Evaluating Concurrent Implementations

4.1 Basic Variants

In accordance with the evaluation of the sequential algorithms, we consider only rank-based and pseudo-random linking strategies combined with all three path compression techniques. Similarly to the existing concurrent DSU implementations, we achieve atomicity by merging trees via `Compare-And-Set (CAS)` primitive.

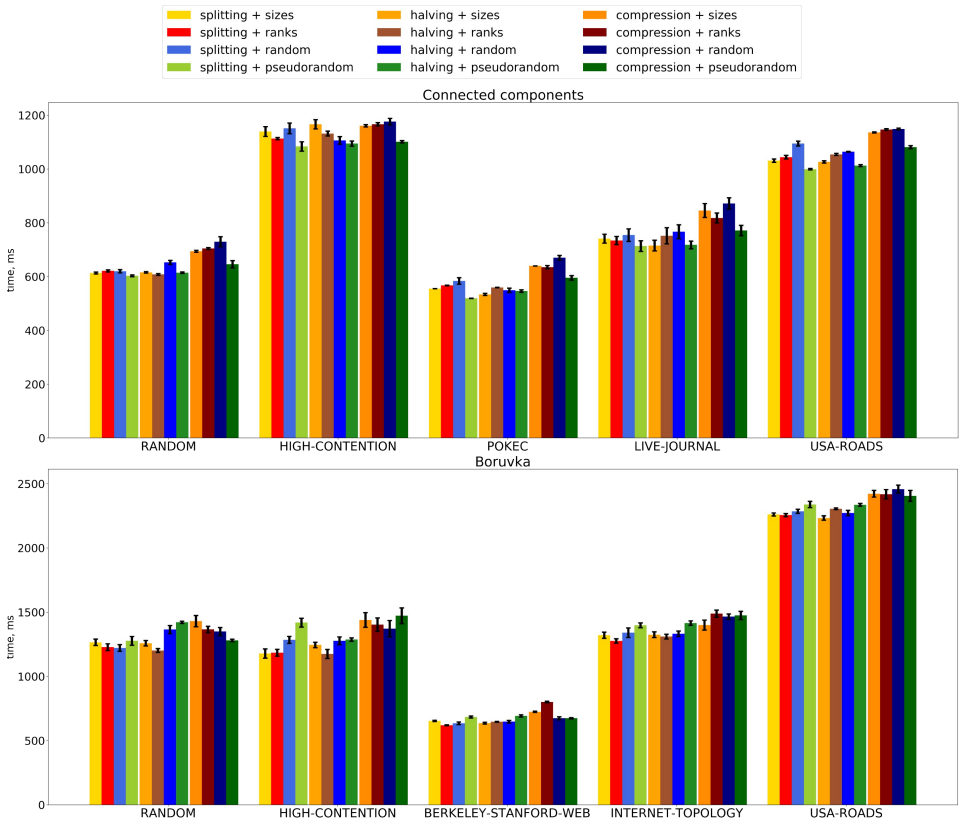


Figure 1 Time comparison for sequential DSU versions with different linking and path compaction strategies on the Intel machine on various input graphs. Lower is better.

Listing 3 contains a pseudo-code for the concurrent version with ranks based on the approach of Anderson and Woll [5]. For strategies with ranks we need to be sure that for a node a rank and a parent can not be updated concurrently, otherwise we can get a non-linearizable behavior, where, for instance, two nodes have parent links pointing to each other violating the forest structure invariant. We can create a structure storing both a parent and a rank and change with *Compare-And-Set* primitive the pointer to a structure, which was proposed by Anderson and Woll, but a faster way would be to store both a rank and a parent in the same register by either dividing it into two parts (i.e. a rank and a parent will have less bits) or by using the same trick as we used in the sequential case, when we stored priorities only for roots. The last technique was chosen since it uses twice less memory. A heuristic that was used for an optimization in the sequential case becomes important for the correctness in the concurrent case.

Figures 2 and 3 present the running time of the basic concurrent implementations for several of the strategies. One may expect that, following the sequential analysis, either rank-based or pseudo-random priorities would behave significantly better, but this is not apparent in the results, what motivates our further investigation. Another natural conjecture, which is apparent empirically, is that splitting and halving compaction strategies have very similar performance (since their cost is similar), while compression is inferior by comparison, due to two path traversals – for finding the root and for path compaction. Thus, we use only splitting heuristic in our next experiments. We also have collected statistics with the numbers of cache misses, which shows that the algorithms with the compression technique get about $\times 7$ more cache misses on loads and almost the same number on stores.

```
1  func Union(u, v) = while (true) {
2    (u, ru) = Find(u)
3    (v, rv) = Find(v)
4    if u == v: return
5    if ru < rv {
6      if CAS(&A[u], u, v): return
7    } else if ru > rv {
8      if CAS(&A[v], v, u): return
9    } else { // ru == rv
10     if u < v && CAS(&A[u], u, v):
11       CAS(&A[v], rv, rv+1); return
12     if u > v && CAS(&A[v], v, u):
13       CAS(&A[u], ru, ru+1); return
14   }
15 }
16 func Find(u): (root, rank) {
17   p = A[u]
18   if isRank(p): return (u, p)
19   (root, rank) = Find(p)
20   if p != root:
21     CAS(&A[u], p, root)
22   return (root, rank)
23 }
24
25 func SameSet(u, v) = while (true) {
26   (u, _) = Find(u); (v, _) = Find(v)
27   if u == v: return true
28   if isRank(A[u]): // still a root?
29     return false
30 }
```

Listing 3 Concurrent DSU with priorities via ranks and the path compression heuristic.

4.2 Optimizations

Path Compaction via Plain Writes. Figure 4 shows the average number of failed CAS operations for different algorithms and thread numbers on the connected component benchmark; we used our local Intel machine to collect these statistics. In worst-case scenarios, only `0.002%` of the total number of CAS invocations fail. Thus, it should be safe to consider that there is almost no contention in practice, and there is no reason to perform several attempts to update parents, which differs from the theoretical worst-case [10].

Since path compaction is a heuristic that influences performance but not correctness, we suggest trying to use simple writes (with or without memory barriers) for updating parent links. What is more, when using writes without memory barriers, we can also use reads without memory barriers at `Find` operations. While this trick is widely-known, our observation is that it does not break the correctness of the DSU algorithm. The only place where we need the reads with memory barriers is the moment when we check that a node is a root. If the check with the parent obtained from a plain read succeeded, then we should re-check using memory barriers. Figure 5 shows the comparison of versions with volatile and plain writes instead of CAS. While the version with volatile memory access does not show significant improvement, the one without memory barriers is faster to up to `40%`. Thus, we consider it useful in almost all further experiments.

Early Recognition. The standard `Union` and `SameSet` implementations perform two `Find` operations on their parameters as the first step. However, if both elements are located in the same set, it could be more efficient to terminate the global operation when both `Find` invocations reach the lowest common ancestor; moreover, it potentially reduces the number of cache misses, which is especially important for NUMA architectures. Similarly, if these two elements are in different sets, it can also be detected during simultaneous climbs – it is guaranteed when the first `Find` reaches the root while the second one stays at an element with greater priority or vice versa. In this situation, `SameSet` can safely return `false`, while `Unite` can link the first root to the current element of the second `Find` climb. Goel et al. used this technique in sequential versions [9], while Jayanti and Tarjan adopted it for the concurrent environment [11]. Since we should know priorities for all nodes, we can not store them only for roots as we have done before for rank-based strategies; thus, we utilize twice more memory. Figure 6 shows that this optimization makes the algorithm faster for some scenarios, while the performance becomes worse or the same on others.

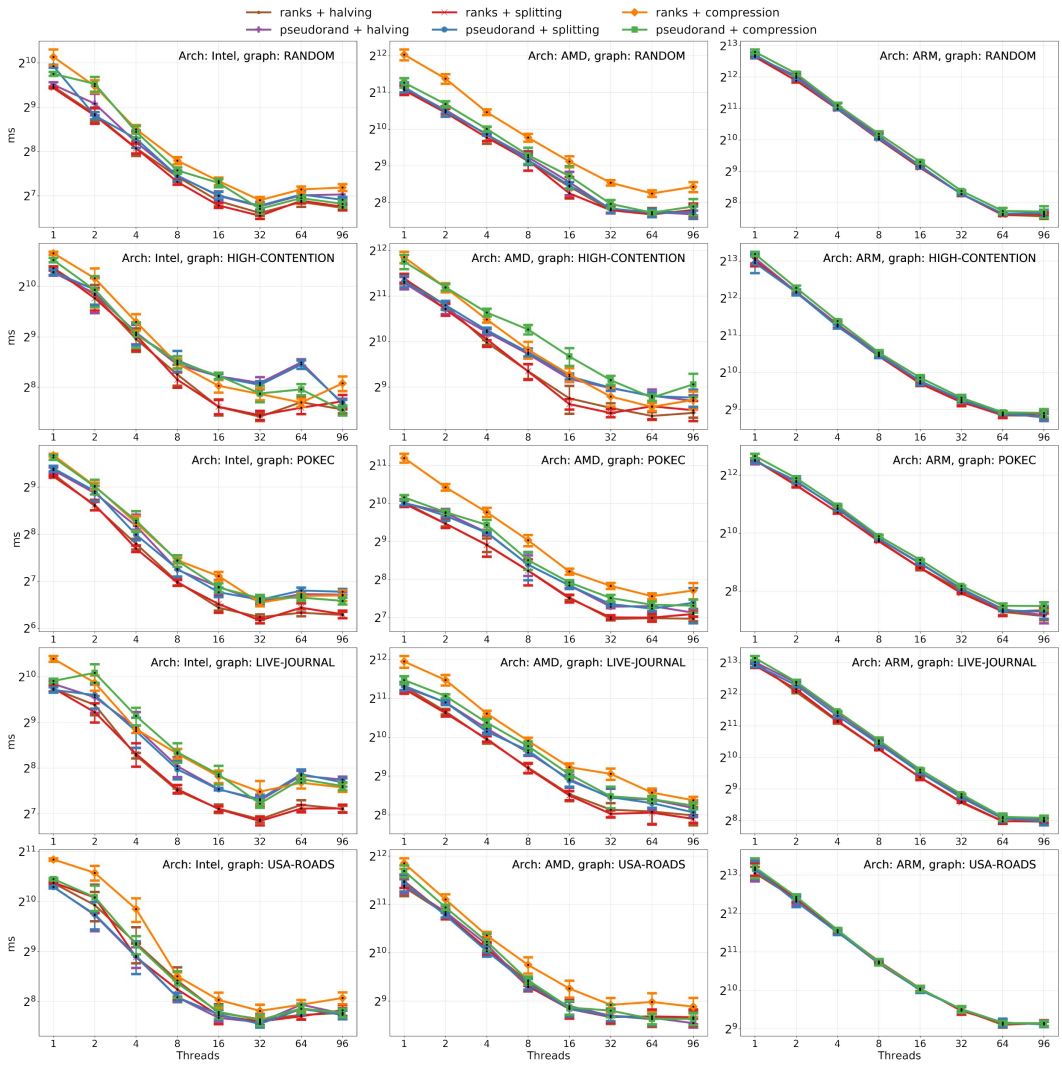


Figure 2 Basic concurrent implementations comparison on the connected components benchmark. Each algorithm has been evaluated on five different graphs (see Table 1) and on Intel, AMD, and ARM platforms. Halving and splitting techniques show the best results, but the compression one is worse on both rank-based and pseudo-random strategies. While the trends on different architectures are similar, there are changes when NUMA effects appear, which can be noticed when the number of threads exceeds the number that one socket can have, thus forcing the algorithm to use several sockets. The versions on Intel and ARM machines do not scale after the moment they need to use two or more NUMA sockets, while the versions on AMD stop scaling only on three sockets. Another observation is that for ARM machines the difference between algorithms is less noticeable, because its processor is slower in comparison to other platforms, while the memory is as fast as in other machines.

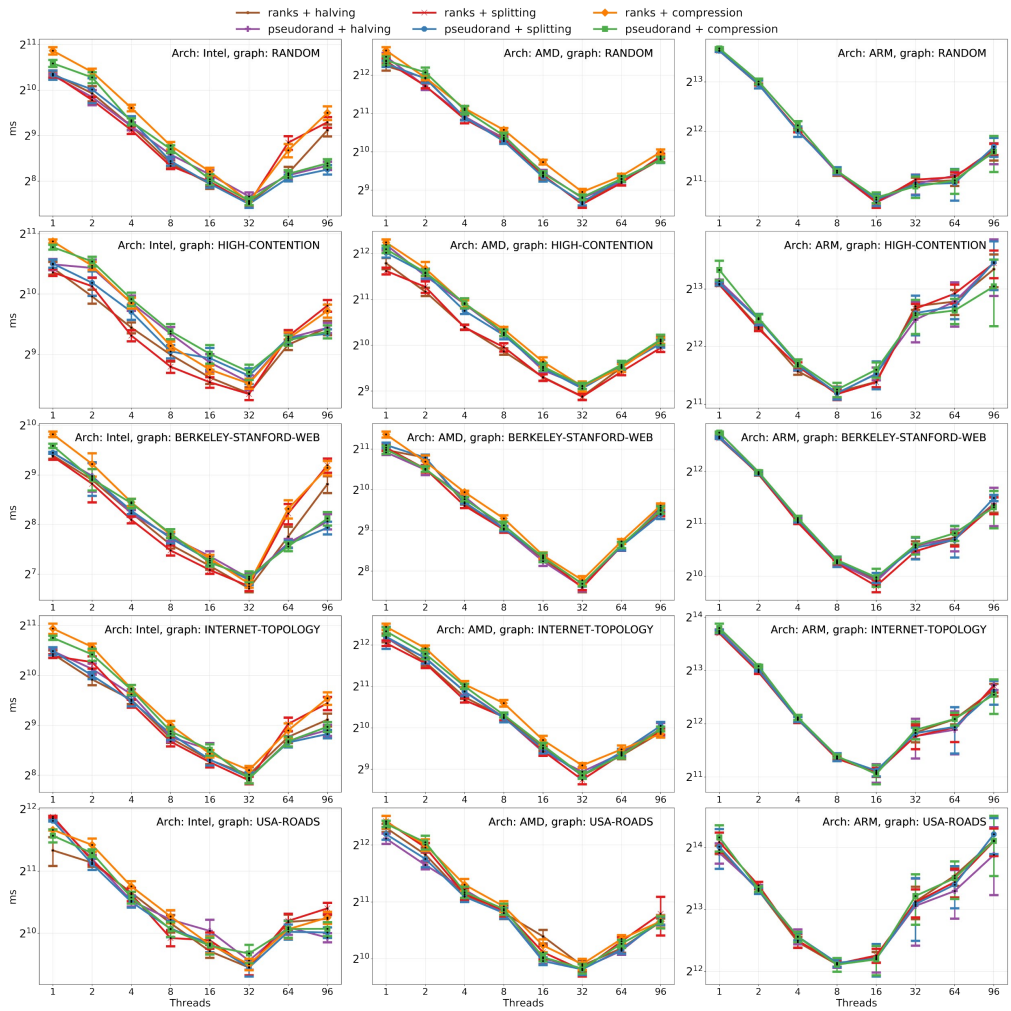


Figure 3 Basic concurrent implementations comparison on the Boruvka's algorithm benchmark. Here the algorithms do not scale well due to internal synchronization; thus, we are mainly interested in the fastest points – on 32 threads for Intel and AMD, on 8 or 16 threads for ARM.

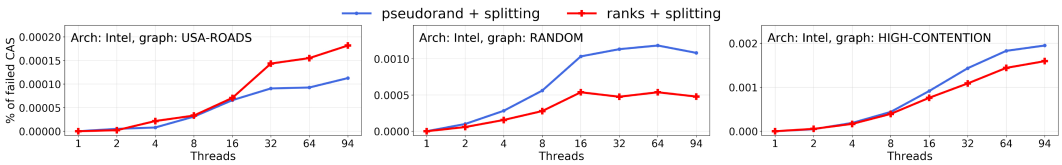


Figure 4 The average number of failed CAS operations on the connected components benchmark (Intel platform). Here we see that almost all CAS operations succeed; thus, in practice, there is no reason to perform several attempts to update parents.

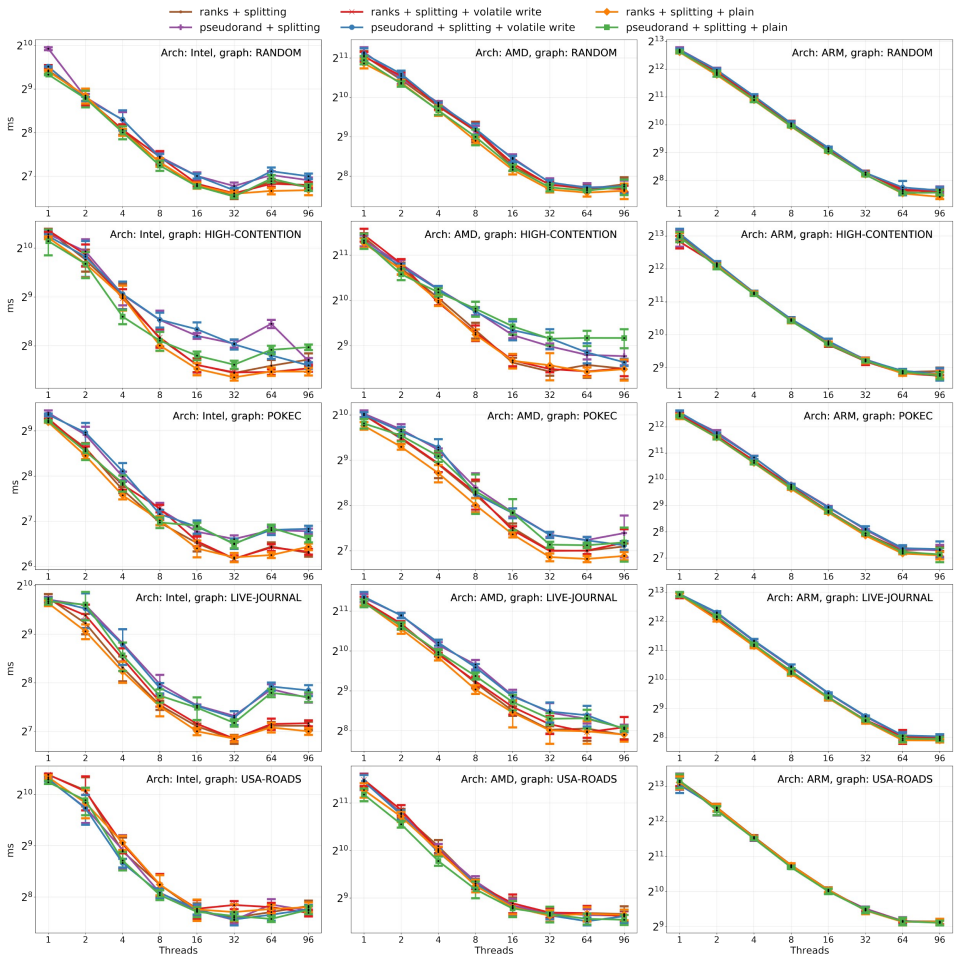


Figure 5 Volatile and plain writes instead of CAS operations to update parents. While volatile writes does not perform better that CAS operations, plain operations give a speedup up to 40%.

```
1  func Unite(u, v) = while (true) {
2    up = parent[u]; vp = parent[v]
3    if u == v or up == vp: return
4    if vp < up:
5      swap(&u, &v); swap(&vp, &up)
6    if u == up:
7      if CAS(&u.parent, u, vp):
8        return true
9    v = parent[up]
10   if up != v:
11     CAS(&u.parent, up, v)
12   u = up
13 }
```

```
14 func SameSet(u, v) = while(true) {
15   up = parent[u]
16   vp = parent[v]
17   if u == v or up == vp:
18     return true
19   if vp < up:
20     swap(&u, &v); swap(&up, &vp)
21   if u == up: return false
22   v = parent[up]
23   if up != v:
24     CAS(&u.parent, up, v)
25   u = up
26 }
```

Listing 4 Concurrent Rem's disjoint-set union algorithm with path splitting.

Immediate Parent Check. Osipov et al. showed that it is likely for two elements to be united having the same parent links [14] (e.g., if the path is almost fully compressed after lots of `Find` invocations). Therefore, they suggested using an *immediate parent check (IPC)* optimization, which examines whether `parent[u]` and `parent[v]` are equal in the beginning of `SameSet` and `Union` operations and if they are, immediately returns. Figure 6 shows that *immediate parent check* significantly improves the performance of all the considered algorithms, especially on the Intel platform. However, the combination of both early recognition and immediate parent check optimizations work worse due to increasing the code complexity on some graphs and architectures. The *IPC* optimization is so efficient on our graphs due to having not many connectivity components; the best version presented in the figure is the one with it but without the *early recognition* optimization.

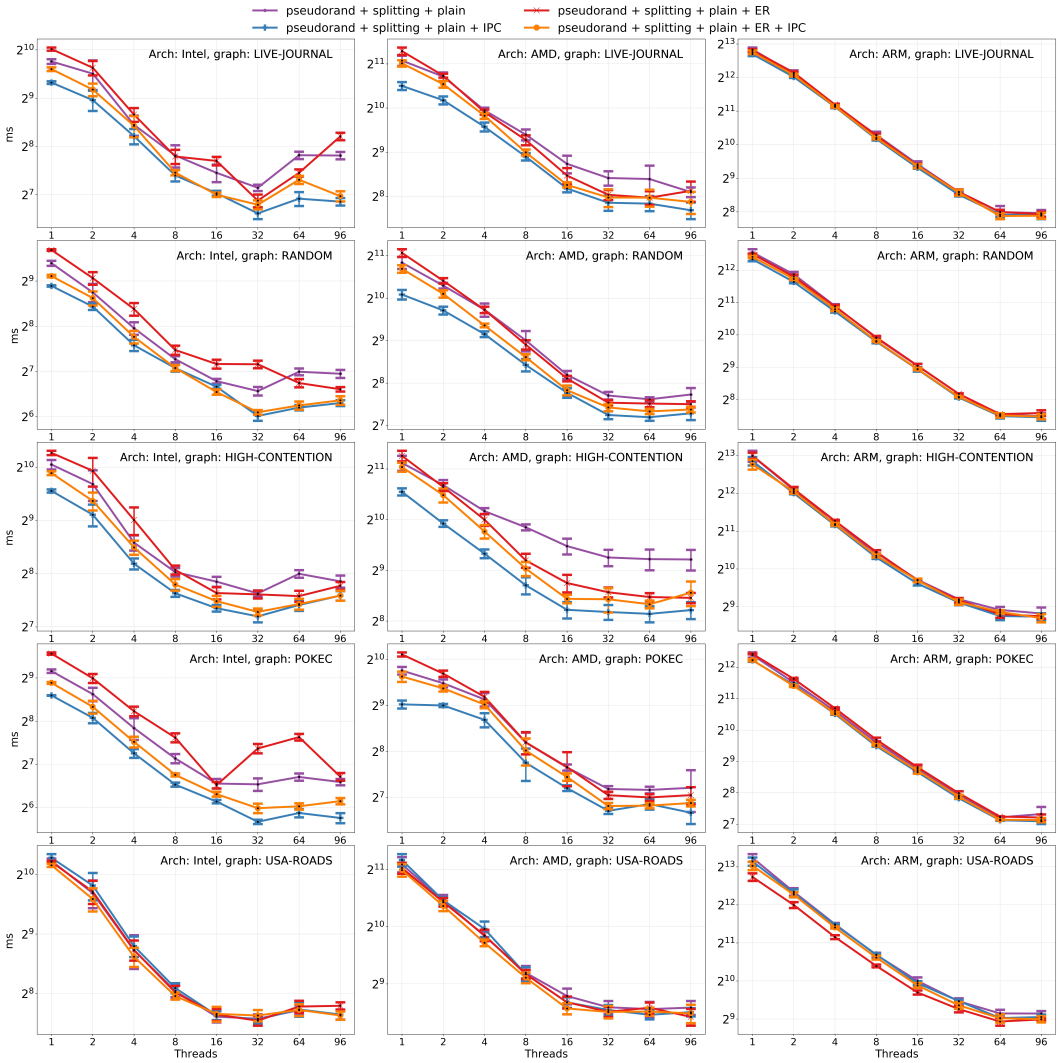


Figure 6 Immediate parent check (IPC) and early recognition (ER) optimizations applied to algorithms with the pseudo-random linking strategy and path compaction via splitting on the connected components benchmark. Here we see that the variant with IPC and without ER wins in all scenarios. We omit the rank-based linking strategy and the Boruvka's algorithm benchmark since the results are similar.

4.3 Rem's algorithm

Rem's algorithm was published in Dijkstra's book "A discipline of programming" [8]. To our knowledge, it is the first known interleaved algorithm, mixing early recognition and a specific linking strategy. The parents are the priorities of nodes. Then we do not need any memory optimizations, because we need nothing but parents. Similarly as in the early recognition algorithm, we perform two searches of roots, concurrently lifting at each step a node with a lower priority. A concurrent version of Rem's algorithm designed by us is presented at listing 4. Rem's algorithm was identified to perform best by Manne and Patwary in the sequential case [15]. While it is the best in the sequential environment, Figure 7 shows that the previously considered algorithms slightly outperform it.

Rem's algorithms are still better than most of the previously examined algorithms, and there are several reasons for it:

- They are based on the *early recognition* optimization.
- Since we need to compare priorities, we already know the parents at each step and can check them for equality having an automatic *immediate parent check* at each step.
- Their priorities are highly correlated with the distance to a root, which is important for *early recognition*. Random priorities never change, ranks change only during `Union` operations, while in Rem's algorithms priorities change after each path compaction, when a distance to the root decreases.

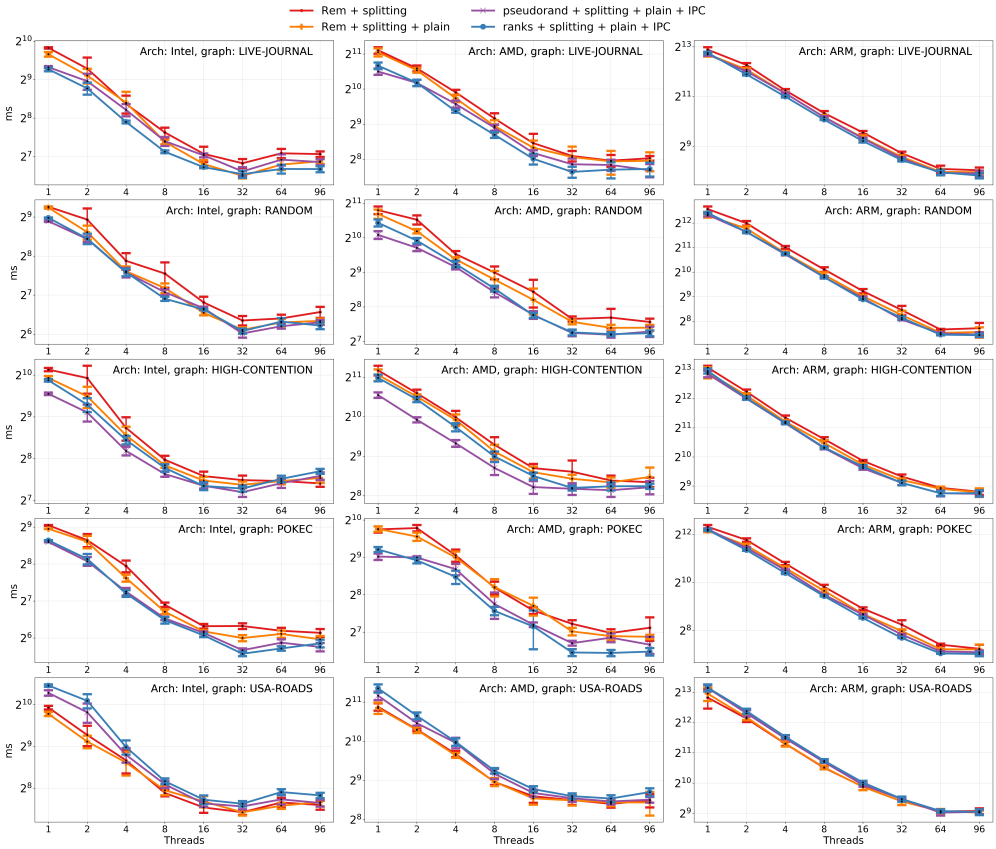


Figure 7 Different variants of Rem's algorithm on the connected components benchmark compared with the previous best algorithms. While Rem's algorithm is the best in the sequential environment, the best of the previously examined ones are slightly better than its concurrent variants.

4.4 Transactional Memory

Since the number of failed CAS operations is imperceptible (see Figure 4), it is reasonable to expect that concurrent operations work with different memory locations almost all the time. Therefore, the union-find problem is potentially a good application for hardware transactional memory (HTM); we expect that almost all transactions will succeed. Instead of trying to use HTM as a fast-path for the concurrent versions, we check whether the best sequential algorithms can be scalable using a coarse-grained lock with the lock elision via HTM optimization [16]. The TM-based algorithms are coupled with all the sequential algorithms we discussed, and with all the corresponding sequential optimizations.

Figure 8 show the connected components benchmark results evaluated on our local Intel server; unfortunately, AMD and ARM do not support transactional memory at this point. The versions with HTM are definitely superior in some of the considered scenarios, being much simpler at the same time. The reason that for `USA-ROADS` and for `HIGH-CONTENTION` graphs on the connected components benchmark they perform better is that these graphs are sparse and operations are unlikely to touch the same memory. Therefore, we conjecture that as general rule algorithms with HTM perform better for sparse graphs. In the extended version of this paper [4] we performed an experiment on the Boruvka benchmark as well; due to more contention versions with plain writes and IPC optimization slightly outperform the HTM-based ones.

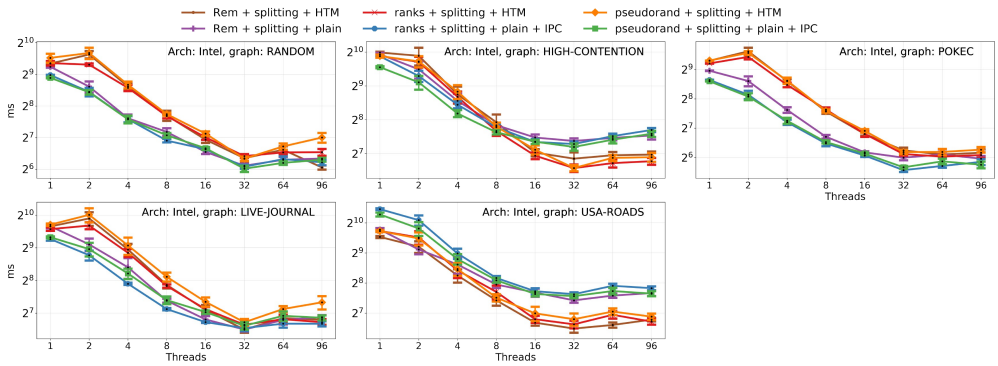


Figure 8 Comparison of lock-based implementations with lock elision via HTM and the best previous algorithms on the connected components benchmark. The versions with HTM are significantly better on two of the considered scenarios and not much worse on other ones, being much simpler at the same time. Graphs where the results are better are sparse, so transactions are less likely to touch the same memory.

5 Discussion

We were the first to perform a thorough study of the practical performance of concurrent DSU implementations, exploring architectures (Intel, AMD, ARM), implementation variants (e.g., baseline, Rem), compaction strategies, as well as optimizations and synchronization techniques (lock-free, HTM). We mention the following salient points.

Memory Transfer Cost. One basic (and predictable) conclusion of our study is that one of the critical factors for the performance of DSU implementations is the amount of memory totally used and accessed per operation, and, in particular, the average number of cache misses per operation. This is natural since the computational cost of this workload is fairly negligible relative to the cost of the memory traversals. Thus, across architectures and

algorithmic variants, implementations that try to minimize memory accesses provide better performance. For instance, this is the case of the pseudo-random priorities optimization, which trades off a trivial amount of extra computation for reduced memory transfer.

Another point is that compression by rank (subtree height) appears to perform particularly well with respect to other methods (in particular, pseudo-random ranks) since it tends to produce higher-quality compressed trees. Notice that this diverges from the theoretical proposals of [10, 11], which avoid ranks due to additional algorithmic complexity.

Synchronization Cost. The second observation regards the synchronization cost of this problem: since most of the overhead of individual operations comes from synchronization cost (e.g., CAS for path compression), it is natural to investigate *relaxing* the strength of the synchronization primitives to improve performance. This is the primary motivation behind replacing CAS with plain or volatile write as an optimization, which does appear to lead to noticeable performance improvements. We note that, in this case, our empirical observations diverge from the theory; for instance, one of the algorithmic proposals of [10] requires a repeated CAS sequence to prove the performance upper bound, whereas our results clearly show that the simple update is sufficient in practice.

Compaction Quality. One of our objectives has been to investigate how the various heuristics for path compaction differ in terms of their practical performance. On this point, we find ourselves unable to adjudicate a clear winner between halving and splitting, which tend to provide similar performance. We do consistently observe that classic path compression leads to worse performance, which is to be expected due to the higher cost.

Scalability. In terms of scalability, we note that many graph algorithms are scalable up to the point where NUMA effects come into play, that is, at the point where threads might be accessing memory outside of their socket. This is especially salient in the case of the Boruvka application, where the cost of merging components across sockets dominates in the multi-socket case. We leave as an exciting future work direction the question of whether provably efficient concurrent NUMA-aware algorithms exist.

Performance Across Architectures. We notice that performance trends are consistent across architectures. Two noticeable differences are the better multi-socket performance of AMD processors (where NUMA effects become detrimental after 3 sockets are used as opposed to just 2 for Intel and ARM), and the fact that the plain write optimizations work better on Intel and AMD. The last observation befalls because ARM processors are slower, while the memory is as fast as in other machines, and we primarily optimize memory accesses.

Hardware Transactional Memory. The outstanding performance of HTM solutions was somewhat surprising for us. We explain this in hindsight by the simple fact that HTM allows the algorithm designer in this case to implement the simplest, most fast-path-efficient variant of the algorithm (in particular, Rem), which avoids most of the average synchronization costs of other techniques and allows more compiler optimizations, while contention is fairly limited in most scenarios.

Performance versus Monetary Cost. Since we ran our benchmarks on cloud instances, we are able to provide some intuition also with respect to the average cost per experimental run. In this case, the clear winner is the ARM machine, which offers significantly cheaper CPU time, at a very high core count.

The Fastest Concurrent Union-Find Algorithm. We conclude that, within the confines of our experimental setup, our optimized concurrent variants with either splitting or halving path compaction strategy and pseudo-random or rank priorities are best across architectures and inputs with an insignificant difference. This finding is perhaps disappointing from the theoretical perspective since we use plain writes instead of several attempts to update parents. Considering that we also use the immediate parent check optimization, it would be interesting to further study the guarantees of the resulting algorithms in the context of concurrent executions, perhaps for common distributions of (non-adversarial) input graphs.

References

1. Amazon Web Services (AWS) - Cloud Computing Services. https://aws.amazon.com/, 2019.
2. JMH - Java Microbenchmark Harness. https://openjdk.java.net/projects/code-tools/jmh/, 2019.
3. Packet Bare Metal - Cloud & Edge Computing Infrastructure Provider. https://www.packet.com/, 2019.
4. Dan Alistarh, Alexander Fedorov, and Nikita Koval. In Search of the Fastest Concurrent Union-Find Algorithm. *arXiv preprint arXiv:1911.06347*, 2019.
5. Richard J. Anderson and Heather Woll. Wait-free Parallel Algorithms for the Union-Find Problem. In *In Proc. 23rd ACM Symposium on Theory of Computing*, pages 370–380, 1994.
6. Otakar Boruvka. O jistém problému minimálním, 1926.
7. George Cybenko, T. G. Allen, and J. E. Polito. Practical parallel Union-Find algorithms for transitive closure and clustering. *International Journal of Parallel Programming*, 17:403–423, 1988.
8. E.W. Dijkstra. *A discipline of programming.* Prentice-Hall series in automatic computation. Prentice-Hall, 1976.
9. Ashish Goel, Sanjeev Khanna, Daniel H. Larkin, and Robert Tarjan. Disjoint Set Union with Randomized Linking. *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1005–1017, January 2014. doi:10.1137/1.9781611973402.75.
10. Siddhartha Jayanti and Robert Tarjan. A Randomized Concurrent Algorithm for Disjoint Set Union. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, pages 75–82, July 2016. doi:10.1145/2933057.2933108.
11. Siddhartha V. Jayanti, Robert E. Tarjan, and Enric Boix-Adserà. Randomized Concurrent Set Union and Generalized Wake-Up. In *PODC*, 2019.
12. Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956.
13. Michael L. Fredman and Michael Saks. The Cell Probe Complexity of Dynamic Data Structures. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washigton, USA*, pages 345–354, January 1989. doi:10.1145/73007.73040.
14. Vitaly Osipov, Peter Sanders, and Johannes Singler. The Filter-Kruskal Minimum Spanning Tree Algorithm. In *ALENEX*, 2009.
15. Md Mostofa Ali Patwary, Jean Blair, and Fredrik Manne. Experiments on union-find algorithms for the disjoint-set data structure. In *International Symposium on Experimental Algorithms*, pages 411–423. Springer, 2010.
16. Ravi Rajwar and James R 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.
17. Robert E. Tarjan and Jan van Leeuwen. Worst-case Analysis of Set Union Algorithms. *J. ACM*, 31:245–281, 1984.
18. Robert Endre Tarjan. Efficiency of a Good But Not Linear Set Union Algorithm. *J. ACM*, 22(2):215–225, April 1975. doi:10.1145/321879.321884.
19. Robert Endre Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of computer and system sciences*, 18(2):110–127, 1979.

On Deterministic Linearizable Set Agreement Objects

Felipe de Azevedo Piovezan
Department of Computer Science, University of Toronto, Canada
felipe@cs.toronto.edu

Vassos Hadzilacos
Department of Computer Science, University of Toronto, Canada
vassos@cs.toronto.edu

Sam Toueg
Department of Computer Science, University of Toronto, Canada
sam@cs.toronto.edu

Abstract

A recent work showed that, for all n and k, there is a *linearizable* (n, k)-set agreement object O_L that is equivalent to the (n, k)-set agreement task [4]: given O_L, it is possible to solve the (n, k)-set agreement task, and given any algorithm that solves the (n, k)-set agreement task (and registers), it is possible to implement O_L. This linearizable object O_L, however, is not deterministic. It turns out that there is also a *deterministic* (n, k)-set agreement object O_D that is equivalent to the (n, k)-set agreement task, but this deterministic object O_D is not linearizable. This raises the question whether there exists a deterministic *and* linearizable (n, k)-set agreement object that is equivalent to the (n, k)-set agreement task. Here we show that in general the answer is no: specifically, we prove that for all $n \geq 4$, every deterministic linearizable $(n, 2)$-set agreement object is strictly stronger than the $(n, 2)$-set agreement task. We prove this by showing that, for all $n \geq 4$, every deterministic and linearizable $(n, 2)$-set agreement object (together with registers) can be used to solve 2-consensus, whereas it is known that the $(n, 2)$-set agreement task cannot do so. For a natural subset of $(n, 2)$-set agreement objects, we prove that this result holds even for $n = 3$.

2012 ACM Subject Classification Theory of computation → Concurrency; Theory of computation → Parallel computing models; Theory of computation → Distributed computing models

Keywords and phrases Asynchronous shared-memory systems, consensus, set agreement, deterministic objects

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.16

Funding This research was partially funded by the Natural Sciences and Engineering Research Council of Canada.

1 Introduction

Consensus is a fundamental problem of distributed computing and set agreement [5] is a well-known generalization of this problem. In the (n, k)-*set agreement task* each of n processes has an input value and must output one of the input values so that there are at most k distinct output values (the special case when $k = 1$ is the n-consensus task) [5, 7, 10]. Researchers have also considered (n, k)-set agreement *objects* [1, 6]. An (n, k)-*set agreement* (SA) *object* is an object that allows up to n processes to invoke a propose operation with some proposal value, such that the following two properties hold: (a) the value returned by the operation has been proposed, and (b) at most k different values are returned. Obviously, every (n, k)-SA object can be used to solve the (n, k)-SA task.

© Felipe de Azevedo Piovezan, Vassos Hadzilacos, and Sam Toueg;
licensed under Creative Commons License CC-BY
23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller; Article No. 16; pp. 16:1–16:15
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Until recently, it was thought that, for $k \geq 2$, (n, k)-SA objects were inherently *not* linearizable,[1] because every *linearizable* (n, k)-SA object imposes constraints that are not required by the (n, k)-SA task. For example, for $k \geq 2$, with any linearizable (n, k)-SA object, the propose operation that is linearized first must return its own proposal value; in contrast, with the (n, k)-set agreement task two processes with distinct inputs are allowed to each output the other's input (i.e., they can "swap" their inputs). In fact, such differences in allowable behaviours between tasks and objects motivated the introduction of generalizations of linearizability [9, 3]. Despite this, it was recently shown that there *does* exist a *linearizable* (n, k)-SA object O_L that is *equivalent to* the (n, k)-set agreement task, in the following sense: (a) given O_L, it is possible to solve the (n, k)-SA task, and (b) given any algorithm that solves the (n, k)-SA task (and registers), it is possible to implement O_L [4].

An object is *deterministic* if, when accessed sequentially, the response of each operation depends uniquely on the sequence of the preceding operations. For $k \geq 2$, the linearizable (n, k)-SA object O_L is *not* deterministic: in a sequential execution where the first operation proposes 1 and the second operation proposes 2, the second operation can return either 1 or 2.

Are (n, k)-SA objects inherently *not* deterministic? The answer is "no". We show that there exists a deterministic (n, k)-SA object O_D that is equivalent to the (n, k)-SA task (see Section 5). For $k \geq 2$, however, O_D is *not* linearizable.

Thus, the (n, k)-SA task has an equivalent linearizable (n, k)-SA object, *and* it also has an equivalent deterministic (n, k)-SA object. This raises the following question: is the (n, k)-SA task equivalent to some (n, k)-SA object that is *both* deterministic and linearizable?

In this paper we explore this question, and answer it in the negative. Specifically, we prove that for $n \geq 4$, no deterministic linearizable $(n, 2)$-SA object is equivalent to the $(n, 2)$-SA task: we do so by showing that, for $n \geq 4$, every deterministic linearizable $(n, 2)$-SA object can be used to solve the 2-consensus task; in contrast, it is known that solving the $(n, 2)$-SA task is not sufficient to solve 2-consensus [1]. This implies that, for $n \geq 4$, every deterministic linearizable $(n, 2)$-SA object is strictly stronger than the $(n, 2)$-SA task. For a natural subset of deterministic linearizable $(n, 2)$-SA objects, we prove that this result holds even for $n = 3$.

2 Sketch of the model and basic definitions

In this paper, we consider distributed systems in which asynchronous processes communicate via shared objects. To limit the number of processes that can concurrently access an object, we consider shared objects with *ports*: to apply an operation on an object, a process chooses one of its ports, invokes the operation at that port, and waits for the response of the operation at that port before invoking another operation. No two operations are allowed to be applied simultaneously on the same port, otherwise the behavior of the object is arbitrary. So an object with n ports can be accessed concurrently by at most n processes. An execution of operations applied to an object is *well formed* if each operation invoked at a port of the object returns before another operation is invoked at that port. Note that a sequential execution of operations on an (n, k)-SA object, i.e., an execution where each operation completes before the next operation starts, is necessarily well formed.

[1] Intuitively, an object is *linearizable* if it behaves as if *all* operations, including concurrent ones, are applied sequentially: each operation appears to take effect instantaneously at some distinct point between its invocation and response [8].

▶ **Definition 1.** *An (n, k)-**set agreement object** is an n-ported object that allows processes to invoke* ProposeSA *operations, each with some proposal value $v \in I$ (where $|I| \geq n$), and the responses of the* ProposeSA *operations satisfy the following two properties in every well-formed execution:*

- Validity*: the value returned by each operation is either its proposal value or the proposal value of some previously invoked* ProposeSA *operation.*
- k-Agreement*: at most k different values are returned by the invocations of* ProposeSA *operations.*

For the rest of this section, O is an (n, k)-set agreement object. As we mentioned in the introduction, O is not necessarily linearizable, but of course it can be accessed sequentially. A sequential execution of operations on O can be modelled by a sequence of triples as follows:

▶ **Definition 2.** *A finite or infinite sequential execution (of operations) on O is a sequence of the form $(p_1, v_1, u_1)(p_2, v_2, u_2)\ldots$, where the i-th operation on O is a* ProposeSA(v_i) *at port p_i of O that returns u_i.*

▶ **Definition 3.** *The **sequential behaviour** of O is the set of* all *possible sequential executions of operations on O.*

Thus, the sequential behaviour of O is a set of sequences of the form $(p_1, v_1, u_1)(p_2, v_2, u_2)\ldots$.

▶ **Notation 4.**

- *If a sequence S is in the sequential behaviour of O, i.e., S is a sequential execution of operations on O, we write $S \in O$.*
- *If $S \in O$ and the* ProposeSA(v) *operation applied immediately after S on port p_i of O can return the value u, we write $S(p_i, v, u) \in O$.*
- *If $S \in O$ but the* ProposeSA(v) *operation applied immediately after S on port p_i of O can **not** return the value u, we write $S(p_i, v, \neq u) \in O$.*

Intuitively, object O is deterministic if for every finite sequential execution of operations S on O the following holds: if S is executed on O, and then a ProposeSA(v) operation is applied on port p of O, the response of that operation is uniquely determined by S, p, and v. More precisely:

▶ **Definition 5.** *O is **deterministic** if for every finite $S \in O$, every port p, and all values v, u, u', if $S(p, v, u) \in O$ and $S(p, v, u') \in O$, then $u = u'$.*

Intuitively, O is linearizable if each operation on O appears to take effect instantaneously between its invocation and its response (even in the presence of concurrent operations) [8].

▶ **Definition 6.** *O is **linearizable** if in every well-formed execution of operations E on O, each operation has a distinct linearization point, between its invocation and its response, such that E appears to be one of the sequential executions of operations $S \in O$, where the operations of S occur in the order of their linearization points in E.*[2]

Because of the above definition, if O is linearizable we can view any well-formed execution of operations on O as one of the sequences $S \in O$.

[2] For simplicity, this definition assumes that all operations of E are complete, i.e., each one has a response. If E contains incomplete operations, one is allowed to discard some incomplete operations and complete the remaining ones by returning some value, before linearizing the now "complete" E.

▶ **Notation 7.** *If an (n,k)-SA object is both deterministic and linearizable, we say that it is an (n,k)-**DLSA object**.*

In the *2-consensus task*, each of two processes has an input value and must output one of the inputs, such that both processes output the same value. Intuitively, we say that O can be used to solve 2-consensus, if two processes can solve the 2-consensus task using registers and copies of O initialized appropriately. More precisely:

▶ **Definition 8.** *We say that O **can be used to solve 2-consensus** if two processes can solve the 2-consensus task using copies of O (each of which may be initialized by applying some finite sequential execution of operations on O) and registers.*[3]

3 Oblivious $(3,2)$-DLSA objects can be used to solve 2-consensus

In this section, we focus on *oblivious* (n,k)-DLSA objects, that is, (n,k)-DLSA objects that behave the same regardless of the port to which operations are applied. We show that for $n \geq 3$, oblivious $(n,2)$-DLSA objects can be used to solve 2-consensus, and thus they are *not* equivalent to the $(n,2)$-SA task.

We first define oblivious (n,k)-DLSA objects more precisely as follows:

▶ **Definition 9.** *Let O be an (n,k)-DLSA object. O is **oblivious** if for all finite sequential executions of operations $S \in O$, ports p_i, p_j, and values u, v, the following holds: If $S(p_i,u,v) \in O$ then $S(p_j,u,v) \in O$.*

Thus, henceforth we omit ports from sequential executions of operations on oblivious (n,k)-SA objects, e.g., instead of $S = (p_1,v_1,u_1)(p_2,v_2,u_2)\ldots(p_k,v_k,u_k)$, we write $S = (v_1,u_1)(v_2,u_2)\ldots(v_k,u_k)$.

By the validity and the k-set agreement properties, it is clear that the first ProposeSA operation applied to an (n,k)-DLSA object must return its own proposal value and the second must return either its own proposal value (allowed only if $k \geq 2$) or the proposal value of the first. Thus,

▶ **Observation 10.** *Let O be an oblivious (n,k)-DLSA object. For all values a, a', b, b', if $(a,a')(b,b') \in O$ then $a = a'$ and $b' \in \{a,b\}$.*

Suppose an oblivious (n,k)-DLSA object O has a finite sequential execution of operations S such that, after S has been applied to O, a process proposing a and another process proposing b can determine the order in which their propose operations were executed. Then O can be used to solve 2-consensus as we now explain.

▶ **Definition 11.** *Let O be an oblivious (n,k)-DLSA object. Let a, b be two (not necessarily distinct) proposal values, and let $S \in O$. Let a', a'', b' be the unique values such that:*

$$\begin{cases} S(b,b')(a,a') \in O \\ S(a,a'') \in O \end{cases}$$

- *If $a' \neq a''$, we say "a **notices** b **after** S", denoted $a \xrightarrow{S} b$.*
- *If $a' = a''$, we say "a **does not notice** b **after** S", denoted $a \not\xrightarrow{S} b$.*

[3] Note that this solution is allowed to use registers, in addition to copies of O; allowing the use of registers in solutions and object implementations is standard.

*If S is the empty sequence, we simply say that a **notices** b, denoted $a \to b$, or a **does not notice** b, denoted $a \not\to b$.*

▶ **Observation 12.** *Let O be an oblivious (n,k)-DLSA object. By Observation 10 and Definition 11, we have that for all distinct proposal values a, b:*

$a \to b$ *if and only if* $(b,b)(a,b) \in O$

$a \not\to b$ *if and only if* $(b,b)(a,a) \in O$

▶ **Definition 13.** *Let O be an oblivious (n,k)-DLSA object. We say that O is **good** if there exist a finite sequential execution $S \in O$ and proposal values a, b such that $a \xrightarrow{S} b$ and $b \xrightarrow{S} a$.*

▶ **Lemma 14.** *Let O be an oblivious (n,k)-DLSA object. If O is good, then O can be used to solve 2-consensus.*

Proof. Let O be an oblivious (n,k)-DLSA object and assume it is good. Since O is good, there exists a finite sequential execution $S \in O$ and two values a, b such that $a \xrightarrow{S} b$ and $b \xrightarrow{S} a$. By the definition of $\xrightarrow{S}$, it follows that there exist a', a'', b', b'' such that $a' \neq a''$, $b' \neq b''$ and:

$$S(a,a')(b,b'') \in O \tag{1}$$

$$S(b,b')(a,a'') \in O \tag{2}$$

To solve 2-consensus, O is first initialized by applying S to it. Two processes, p and q, can now solve 2-consensus using this initialized O and two registers, R_p (written only by p) and R_q (written only by q), by executing the following algorithm. We assume that p and q use *different* ports when accessing O, so their access to O is guaranteed to be well-formed.

```
PROPOSE(v_p) by process p
1  R_p := v_p
2  ret := PROPOSESA(a) on a port of O
3  if ret = a'
4      decide R_p
5  else // ret = a''
6      decide R_q
```

```
PROPOSE(v_q) by process q
1  R_q := v_q
2  ret := PROPOSESA(b) on a port of O
3  if ret = b'
4      decide R_q
5  else // ret = b''
6      decide R_p
```

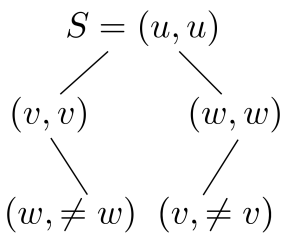
Since O is linearizable (and it is accessed in a well-formed manner), we can regard the PROPOSESA operations of p and q as atomic, and thus the PROPOSESA operations are totally ordered. From the algorithm and (1), (2) it is clear that each process decides the value it reads from the register it writes if it is the first to execute a PROPOSESA operation; otherwise, it decides the value it reads from the register written by the other process. Therefore agreement and validity hold provided that the process that reads the register written by the other process (in its Line 6) does so after the other process has written into that register (in its Line 1). To see that if process p reads register R_q, p does so after process q has written into R_q, note that process p reads R_q if and only if its PROPOSESA operation does not return a'. By (1) and (2), this means that q executed its PROPOSESA operation (in Line 2) before p executes its ProposeSA operation; and therefore q writes into R_q (in its Line 1) before p reads R_q (in its Line 6). A similar argument applies to q. ◀

We will now prove that oblivious $(3,2)$-DLSA objects that are *not* good have some other properties that can also be exploited to solve 2-consensus.

▶ **Lemma 15.** *Let O be an oblivious $(3,2)$-DLSA object. If O is not good, the following holds. For all distinct proposal values u, v, w, if $v \not\to u$ then $w \to u$.*

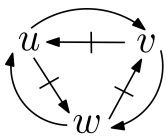
Proof. Let O be an oblivious $(3,2)$-DLSA object and assume it is not good. Suppose $v \not\to u$. By Observation 12, $(u,u)(v,v) \in O$. Furthermore, if w is proposed after that, the response of this operation cannot be w, by the 2-agreement property (see the left branch of the figure below). So $(u,u)(v,v)(w,{\neq}w) \in O$.

Assume for contradiction that $w \not\to u$. By Observation 12 $(u,u)(w,w) \in O$. Furthermore, if v is proposed after that, the response of this operation cannot be v, by the 2-agreement property (right branch of the figure below). So $(u,u)(w,w)(v,{\neq}v) \in O$.



Let $S = (u,u)$. Since $S(v,v) \in O$ and $S(w,w)(v,{\neq}v) \in O$, $v \xrightarrow{S} w$. Similarly, $w \xrightarrow{S} v$, and so O is good, a contradiction. ◀

▶ **Lemma 16.** *Let O be an oblivious $(3,2)$-DLSA object. If O is not good, the following holds. For all distinct proposal values u, v, w, if $v \not\to u$ then all of the relationships illustrated in the diagram below hold:*



Proof. Let O be an oblivious $(3,2)$-DLSA object and assume it is not good. Suppose $v \not\to u$. We prove each of the arrows in turn:

- $w \to u$: Since $v \not\to u$, Lemma 15 implies $w \to u$.
- $u \not\to w$: Because O is not good and $w \to u$, it follows that $u \not\to w$.
- $v \to w$: Since $u \not\to w$, Lemma 15 implies $v \to w$.
- $w \not\to v$: Because O is not good and $v \to w$, it follows that $w \not\to v$.
- $u \to v$: Since $w \not\to v$, Lemma 15 implies $u \to v$. ◀

▶ **Lemma 17.** *Let O be an oblivious $(3,2)$-DLSA object. If O cannot be used to solve 2-consensus, the following holds. For all distinct proposal values u, v, w, if $u \to v$ then there exists $x \neq w$ such that:*

$$(v,v)(u,v)(w,x) \in O$$
$$(u,u)(v,v)(w,x) \in O$$

Proof. Let O be an oblivious $(3,2)$-DLSA object and assume it cannot be used to solve 2-consensus. So, by Lemma 14, O is not good.

Assume $u \to v$, thus, by Observation 12, $(v,v)(u,v) \in O$. By the definition of DLSA objects, there exists x such that:

$$(v,v)(u,v)(w,x) \in O \tag{3}$$

Since O is not good, $u \to v$ implies $v \not\to u$. By Observation 12, $(u,u)(v,v) \in O$, and so there exists x' such that:

$$(u,u)(v,v)(w,x') \in O \tag{4}$$

The 2-agreement property implies that $x' \in \{u,v\}$, thus $w \neq x'$.

Because $v \not\to u$, Lemma 16 implies $w \not\to v$. By Observation 12, $(v,v)(w,w) \in O$. Thus, by the 2-agreement property:

$$(v,v)(w,w)(u,{\neq}u) \in O \tag{5}$$

Our goal is to show that $x = x'$. Suppose, for contradiction, that $x \neq x'$.

Two processes, p and q, can solve 2-consensus using O and two registers, R_p (written only by p) and R_q (written only by q), by executing the following algorithm. We assume that p and q use *different* ports when accessing O, so their access to O is guaranteed to be well-formed.

```
PROPOSE(v_p) by process p
R_p := v_p
ret := PROPOSESA(u) on a port of O
if ret = u
    decide R_p
else
    decide R_q
```

```
PROPOSE(v_q) by process q
R_q := v_q
PROPOSESA(v) on a port of O
ret := PROPOSESA(w) on a port of O
if ret = x'
    decide R_p
else
    decide R_q
```

To see why this algorithm is correct, first notice that the possible interleavings of the three PROPOSESA operations by p and q are:

- v, u, w (process q, then p, then q)
- u, v, w (process p, then q, then q)
- v, w, u (process q, then q, then p)

The values returned by O in these three cases are given in (3), (4) and (5), respectively. We now argue that the algorithm is correct. Termination is obvious since the algorithm is wait free.

Validity trivially holds if a process decides the value in the register it writes. So consider the cases where each process decides the value contained in the register of the other.

- Process p decides R_q iff its PROPOSESA(u) operation returns a value different from u, which only happens in (3) and (5). In both cases, q has previously proposed v, in which case q has previously written its proposal into R_q.
- Process q decides R_p iff its PROPOSESA(w) operation returns x', which only happens in (4). In that case, p has previously proposed u, in which case p has previously written its proposal into R_p.

In both cases, validity holds.

Agreement also follows from (3), (4) and (5). If only one process decides, then agreement holds trivially. So suppose both processes decide.

- If p decides R_p, then its PROPOSESA(u) operation returned u, which only happens in (4). So q receives x' as the response of its PROPOSESA(w) operation, and thus q also decides R_p.

- If p decides R_q, then its PROPOSESA(u) operation returned a value different from u, which only happens in (3) and (5). In those cases, q receives $x \neq x'$ (in (3)) or $w \neq x'$ (in (5)) as the response of its PROPOSESA(w) operation. Either way, the value q receives is different from x', and thus q also decides R_q.

In both cases, agreement holds.

We have shown that O can be used to solve 2-consensus, a contradiction. Therefore, $x = x'$, as desired. ◀

▶ **Lemma 18.** *Let O be an oblivious $(3, 2)$-DLSA object. If O cannot be used to solve 2-consensus, the following holds. For all distinct proposal values u, v, w, if $u \to v$ then $(v, v)(w, w)(u, v) \in O$.*

Proof. Let O be an oblivious $(3, 2)$-DLSA object and assume it cannot be used to solve 2-consensus. So, by Lemma 14, O is not good.

Assume $u \to v$. By Lemma 16, $w \not\to v$, therefore $(v, v)(w, w) \in O$ (Observation 12). Also, there exists x' such that $(v, v)(w, w)(u, x') \in O$. By the 2-agreement property, $x' \in \{v, w\}$. If $x' = v$, we are done, so assume, for contradiction, that $x' = w$, i.e., $(v, v)(w, w)(u, w) \in O$(*).

Since $u \to v$, by Lemma 17, there exists $x \neq w$ such that $(v, v)(u, v)(w, x) \in O$, in other words, $(v, v)(u, v)(w, \neq w) \in O$ (**). Let $S = (v, v)$; then (*) and (**) imply that $u \xrightarrow{S} w$ and $w \xrightarrow{S} u$. Thus O is good, a contradiction. So it must be that $x' = v$. ◀

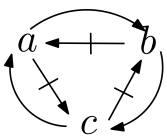
We are now ready to prove the following theorem:

▶ **Theorem 19.** *Every oblivious $(3, 2)$-DLSA object can be used to solve 2-consensus.*

Proof. Let O be an oblivious $(3, 2)$-DLSA object and let a, b, c be three distinct proposal values of O. Suppose, for contradiction, that O cannot be used to solve 2-consensus. By Lemma 14, O is not good.

Because O is not good, either $a \not\to b$ or $b \not\to a$. Without loss of generality, assume $b \not\to a$. By Lemma 16, we have:

▶ **Observation 20.** *All of the relationships among a, b, and c illustrated in the diagram below hold:*



By Observation 20, $c \to a$. Applying Lemma 18 with $c = u$, $a = v$ and $b = w$, it follows that $(a, a)(b, b)(c, a) \in O$. By the 2-agreement property, if we then propose b, the return value of this operation cannot be c. Therefore:

$$(a, a)(b, b)(c, a)(b, \neq c) \in O \tag{6}$$

By Observation 20, $b \to c$. Applying Lemma 18 with $b = u$, $c = v$ and $a = w$, it follows that $(c, c)(a, a)(b, c) \in O$ (*). Furthermore, since $c \to a$, by Lemma 17 with $c = u$, $a = v$ and $b = w$, there exists $x \neq b$ such that:

$$(a, a)(c, a)(b, x) \in O$$
$$(c, c)(a, a)(b, x) \in O$$

By (*) and the fact that O is deterministic, $x = c$. Thus, $(a, a)(c, a)(b, c) \in O$. Now let y be such that

$$(a, a)(c, a)(b, c)(b, y) \in O$$

By the 2-agreement property, $y \in \{a, c\}$. If $y = a$, for $S = (a, a)(c, a)$, we have $S(b, c)(b, a) \in O$. Therefore $b \xrightarrow{S} b$, contradicting the fact that O is not good. Thus $y = c$ and

$$(a, a)(c, a)(b, c)(b, c) \in O \tag{7}$$

To solve 2-consensus, O is first initialized by applying $S = (a, a)$ to it. Two processes, p and q, can now solve 2-consensus using this initialized O and two registers, R_p (written only by p) and R_q (written only by q), by executing the following algorithm. We assume that p and q use *different* ports when accessing O, so their access to O is guaranteed to be well-formed.

```
PROPOSE(v_p) by process p
R_p := v_p
ret := PROPOSESA(b) on a port of O
if ret = b
    decide R_p
else
    decide R_q
```

```
PROPOSE(v_q) by process q
R_q := v_q
PROPOSESA(c) on a port of O
ret := PROPOSESA(b) on a port of O
if ret ≠ c
    decide R_p
else
    decide R_q
```

To see why this algorithm is correct, first notice that the possible interleavings of the three PROPOSESA operations by p and q are:

- b, c, b (process p, then q, then q)
- c, b, b (process q, then p, then q)
- c, b, b (process q, then q, then p)

Notice that the second and third interleavings have the same sequence of proposal values. The values returned by O in each of those cases is given in (6) and (7), under the assumption that the object is initialized by applying (a, a) to it. We now argue that the algorithm is correct; the arguments are similar to those of Lemma 17. Termination is obvious since the algorithm is wait free.

Validity trivially holds if a process decides the value in the register it writes. So consider the cases where each process decides the value contained in the register of the other.

- Process p decides R_q iff its PROPOSESA(b) operation returns a value different from b, which only happens in (7). In that case, q has previously proposed c, in which case q has previously written its proposal into R_q.
- Process q decides R_p iff its PROPOSESA(b) operation returns a value different from c, which only happens in (6). In that case, p has previously proposed b, in which case p has previously written its proposal into R_p.

In both cases, validity holds.

Agreement also follows from the properties of O's specification that we derived above. If only one process decides, then agreement holds trivially. So suppose both processes decide.

- If p decides R_p, then its PROPOSESA(b) operation returned b, which only happens in (6). So q receives a value different from c as the response of its PROPOSESA(b) operation, and thus q also decides R_p.
- If p decides R_q, then its PROPOSESA(b) operation returned a value different from b, which only happens in (7). So q receives c as the response of its PROPOSESA(b) operation, and thus q also decides R_q.

In both cases, agreement holds.

Under the assumption that O cannot be used to solve 2-consensus, we have shown that O can be used to solve 2-consensus, a contradiction. We conclude that O can indeed be used to solve 2-consensus. ◀

▶ **Corollary 21.** *For all $n \geq 3$, every oblivious $(n, 2)$-DLSA object can be used to solve 2-consensus.*

▶ **Theorem 22.** *For all $n \geq 3$, there is no oblivious $(n, 2)$-DLSA object that is equivalent to the $(n, 2)$-SA task.*

Proof. It is known that it is impossible to solve the 2-consensus task using an arbitrary solution to the $(n, 2)$-SA task (and registers) [1]. The theorem now follows immediately by Corollary 21. ◀

4 $(4, 2)$-DLSA objects can be used to solve 2-consensus

We now prove that for $n \geq 4$, $(n, 2)$-DLSA objects can be used to solve 2-consensus, which implies that they are not equivalent to the $(n, 2)$-SA task.

Recall that $(4, 2)$-DLSA objects accept at least 4 different values as proposal values, and have 4 ports to which PROPOSESA operations may be applied. The proof that such objects can be used to solve 2-consensus is simpler than the oblivious $(3, 2)$-DLSA case.

By the validity and the k-set agreement properties, it is clear that the first PROPOSESA operation applied to an (n, k)-DLSA object must return its own proposal value and the second must return either its own proposal value (allowed only if $k \geq 2$) or the proposal value of the first. Thus,

▶ **Observation 23.** *Let O be an (n, k)-DLSA object. For any two (not necessarily distinct) ports p_1, p_2 and for all values a, a', b, b', if $(p_1, a, a')(p_2, b, b') \in O$ then $a = a'$ and $b' \in \{a, b\}$.*

Suppose an (n, k)-DLSA object O has a finite sequential execution of operations S such that, after S has been applied to O, a process proposing a to some port p_i and another process proposing b to some port p_j can determine the order in which their propose operations were executed. Then O can be used to solve 2-consensus as we now explain.

▶ **Definition 24.** *Let O be an (n, k)-DLSA object. Let a, b be two (not necessarily distinct) proposal values, let p_i, p_j be two (not necessarily distinct) ports, and let $S \in O$. Let a', a'', b' be the unique values such that:*

$$\begin{cases} S(p_j, b, b')(p_i, a, a') \in O \\ S(p_i, a, a'') \in O \end{cases}$$

- *If $a' \neq a''$, we say that (p_i, a) **notices** (p_j, b) **after** S, denoted $(p_i, a) \xrightarrow{S} (p_j, b)$.*
- *If $a' = a''$, we say that (p_i, a) **does not notice** (p_j, b) **after** S, denoted $(p_i, a) \not\xrightarrow{S} (p_j, b)$.*

If S is the empty sequence, we simply say (p_i, a) ***notices*** *(p_j, b), denoted $(p_i, a) \to (p_j, b)$, or (p_i, a)* ***does not notice*** *(p_j, b), denoted $(p_i, a) \not\to (p_j, b)$.*

▶ **Observation 25.** *Let O be an (n, k)-DLSA object. By Observation 23 and Definition 24, we have that for all (not necessarily distinct) ports p_i, p_j and for all distinct proposal values a, b:*

$$(p_i, a) \to (p_j, b) \text{ if and only if } (p_j, b, b)(p_i, a, b) \in O$$
$$(p_i, a) \not\to (p_j, b) \text{ if and only if } (p_j, b, b)(p_i, a, a) \in O$$

▶ **Definition 26.** *Let O be an (n, k)-DLSA object. We say that O is* ***good*** *if there exist a finite sequential execution $S \in O$, two proposal values a, b, and two* distinct *ports p_i, p_j such that $(p_i, a) \xrightarrow{S} (p_j, b)$ and $(p_j, b) \xrightarrow{S} (p_i, a)$.*

▶ **Lemma 27.** *Let O be an (n, k)-DLSA object. If O is good, then O can be used to solve 2-consensus.*

Proof. Let O be a (n, k)-DLSA object and assume it is good. Since O is good, there exists a finite sequential execution $S \in O$ two values a, b and two distinct ports p_i, p_j such that $(p_i, a) \xrightarrow{S} (p_j, b)$ and $(p_j, b) \xrightarrow{S} (p_i, a)$.

By the definition of $\xrightarrow{S}$, it follows that there exist a', a'', b', b'' such that $a' \neq a''$, $b' \neq b''$ and:

$$S(p_i, a, a')(p_j, b, b'') \in O \tag{8}$$
$$S(p_j, b, b')(p_i, a, a'') \in O \tag{9}$$

To solve 2-consensus, O is first initialized by applying S to it. Two processes, p and q, can now solve 2-consensus using this initialized O and two registers, R_p (written only by p) and R_q (written only by q), by executing the following algorithm. Recall that p_i and p_j are distinct, so the two processes apply their ProposeSA operations to different ports, ensuring that there are no concurrent operations applied to any port.

Propose(v_p) by process p	Propose(v_q) by process q
1 $R_p \coloneqq v_p$	1 $R_q \coloneqq v_q$
2 $ret \coloneqq$ ProposeSA(a) on port p_i of O	2 $ret \coloneqq$ ProposeSA(b) on port p_j of O
3 **if** $ret = a'$	3 **if** $ret = b'$
4 decide R_p	4 decide R_q
5 **else** // $ret = a''$	5 **else** // $ret = b''$
6 decide R_q	6 decide R_p

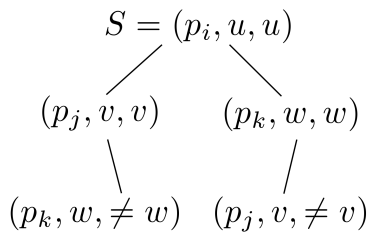
From the algorithm and (8), (9) it is clear that each process decides the value it reads from the register it writes if it is the first to execute a ProposeSA operation; otherwise, it decides the value it reads from the register written by the other process. Therefore agreement and validity hold provided that the process that reads the register written by the other process (in its Line 6) does so after the other process has written into that register (in its Line 1). To see that if process p reads register R_q, p does so after process q has written into R_q, note that process p reads R_q if and only if its ProposeSA operation does not return a'. By (8) and (9), this means that q executed its ProposeSA operation (in Line 2) before p executes its ProposeSA operation; and therefore q writes into R_q (in its Line 1) before p reads R_q (in its Line 6). A similar argument applies to q. ◀

We will now prove that $(4,2)$-DLSA objects that are *not* good have some other properties that can also be exploited to solve 2-consensus.

▶ **Lemma 28.** *Let O be a $(4,2)$-DLSA object. If O is not good, the following holds. For all distinct proposal values u, v, w and all ports p_i, p_j, p_k such that $p_j \neq p_k$, if $(p_j, v) \not\to (p_i, u)$ then $(p_k, w) \to (p_i, u)$.*

Proof. Let O be a $(4,2)$-DLSA object and assume that it is not good. Suppose $(p_j, v) \not\to (p_i, u)$. By Observation 25, $(p_i, u, u)(p_j, v, v) \in O$. Furthermore, if w is proposed after that on port p_k, the response of this operation cannot be w, by the 2-agreement property (see the left branch of the figure below). So $(p_i, u, u)(p_j, v, v)(p_k, w, \neq w) \in O$.

Now assume, for contradiction, that $(p_k, w) \not\to (p_i, u)$. Then, by Observation 25, $(p_i, u, u)(p_k, w, w) \in O$. Furthermore, if v is proposed after that on port p_j, the response of this operation cannot be v, by the 2-agreement property (right branch of the figure below). So $(p_i, u, u)(p_k, w, w)(p_j, v, \neq v) \in O$.



Let $S = (p_i, u, u)$. Since $S(p_j, v, v) \in O$ and $S(p_k, w, w)(p_j, v, \neq v) \in O$, $(p_j, v) \xrightarrow{S} (p_k, w)$. Similarly, $(p_k, w) \xrightarrow{S} (p_j, v)$, and so O is good, a contradiction. ◀

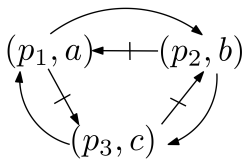
We are now ready to prove the following theorem:

▶ **Theorem 29.** *Every $(4,2)$-DLSA object can be used to solve 2-consensus.*

Proof. Let O be a $(4,2)$-DLSA object, let p_1, p_2, p_3, p_4 be distinct ports and let a, b, c, d be distinct proposal values. Suppose, for contradiction, that O cannot be used to solve 2-consensus. By Lemma 27, O is not good.

Because O is not good, either $(p_2, b) \not\to (p_1, a)$ or $(p_1, a) \not\to (p_2, b)$. Without loss of generality, assume that $(p_2, b) \not\to (p_1, a)$ (*).

▶ **Claim 29.1.** *All of the relationships among (p_1, a), (p_2, b), and (p_3, c) illustrated in the diagram below hold:*

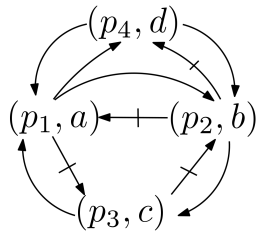


Proof. We prove each of the relationships in turn:

- $(p_2, b) \not\to (p_1, a)$: this is (*).
- $(p_3, c) \to (p_1, a)$: since $(p_2, b) \not\to (p_1, a)$, Lemma 28 implies $(p_3, c) \to (p_1, a)$.
- $(p_1, a) \not\to (p_3, c)$: since O is not good and $(p_3, c) \to (p_1, a)$, we have $(p_1, a) \not\to (p_3, c)$.
- $(p_2, b) \to (p_3, c)$: since $(p_1, a) \not\to (p_3, c)$, Lemma 28 implies $(p_2, b) \to (p_3, c)$.

- $(p_3, c) \not\to (p_2, b)$: since O is not good and $(p_2, b) \to (p_3, c)$, we have $(p_3, c) \not\to (p_2, b)$.
- $(p_1, a) \to (p_2, b)$: since $(p_3, c) \not\to (p_2, b)$, Lemma 28 implies $(p_1, a) \to (p_2, b)$. ◀

Now consider what happens to port p_4 and value d:



- $(p_4, d) \to (p_2, b)$: by Claim 29.1, $(p_3, c) \not\to (p_2, b)$. Then, by Lemma 28, $(p_4, d) \to (p_2, b)$.
- $(p_2, b) \not\to (p_4, d)$: because $(p_4, d) \to (p_2, b)$ and because O is not good, it must be the case that $(p_2, b) \not\to (p_4, d)$.
- $(p_1, a) \to (p_4, d)$: because $(p_2, b) \not\to (p_4, d)$, by Lemma 28 it follows that $(p_1, a) \to (p_4, d)$.
- $(p_4, d) \to (p_1, a)$: by Claim 29.1, $(p_2, b) \not\to (p_1, a)$. Then, by Lemma 28, $(p_4, d) \to (p_1, a)$.

Since $(p_1, a) \to (p_4, d)$ and $(p_4, d) \to (p_1, a)$, O is good, a contradiction. Therefore O can be used to solve 2-consensus, as desired. ◀

▶ **Corollary 30.** *For all $n \geq 4$, every $(n, 2)$-DLSA object can be used to solve 2-consensus.*

▶ **Theorem 31.** *For all $n \geq 4$, there is no $(n, 2)$-DLSA object that is equivalent to the $(n, 2)$-SA task.*

Proof. It is known that it is impossible to solve the 2-consensus task using an arbitrary solution to the $(n, 2)$-SA task (and registers) [1]. The theorem now follows immediately by Corollary 30. ◀

5 Existence of a deterministic (n, k)-SA object equivalent to the (n, k)-SA task

In this section we prove that for all n and k, there is a deterministic (n, k)-set agreement object O_D that is equivalent to the (n, k)-set agreement task. This object is not linearizable, but its behaviour is deterministic when it is accessed sequentially (so concurrency is its only source of non-determinism).

Fix any (n, k)-SA object that is equivalent to the (n, k)-SA task. As we mentioned in the introduction, such objects exist [1, 6]; for concreteness we choose here the linearizable (n, k)-SA object O_L defined in [4]. Since O_L is equivalent to the (n, k)-SA task: (a) given O_L, it is possible to solve the (n, k)-SA task, and (b) given any algorithm that solves the (n, k)-SA task (and registers), it is possible to implement O_L. The simple algorithm of Figure 1 shows how to use O_L and a shared register R (initialized to $\perp$) to implement a *deterministic* (n, k)-set agreement object O_D:

It is easy to see that O_D behaves as follows:

- When accessed sequentially, O_D behaves deterministically: every ProposeSA operation on O_D returns the proposal value of the *first* ProposeSA operation on O_D. In other words, in sequential executions, O_D behaves like a consensus object.
- In all executions, O_D respects validity and k-agreement.

Thus, O_D is indeed a deterministic (n, k)-set agreement object.

```
ProposeSA(v) on port p_i of O_D
temp := R
if temp = ⊥
    temp := ProposeSA(v) on port p_i of O_L
    R := temp
decide temp
```

Figure 1 Implementation of O_D using O_L and registers.

▶ **Theorem 32.** *For all $n \geq k \geq 1$, the deterministic (n, k)-SA object O_D is equivalent to the (n, k)-SA task.*

Proof. Clearly O_D can be used to solve the (n, k)-SA task. It remains to show that given any algorithm $\mathcal{A}_{n,k}$ that solves the (n, k)-SA task (and registers), it is possible to implement O_D. Since O_L is equivalent to the (n, k)-SA task, it can be implemented using $\mathcal{A}_{n,k}$ (and registers). Plugging this implementation of O_L in the algorithm of Figure 1, gives an implementation of O_D that uses $\mathcal{A}_{n,k}$ (and registers). ◀

Note that for $k \geq 2$, the deterministic object O_D is *not* linearizable, even though the object O_L is: a concurrent execution of two ProposeSA operations on O_D with different proposal values may result in each operation returning its own proposal, but this behaviour is not possible in any *sequential* execution of these operations on O_D.

6 Remark on object initialization

To obtain our results, we gave several algorithms that solve 2-consensus using (a single copy of) some (n, k)-DLSA object O that was initialized by applying some sequence of operations. The reader may ask whether these results still apply if algorithms solving 2-consensus are required to use only *un*initialized copies of O. The answer is yes. This is because, by a result of [2], the ability to solve 2-consensus using an object does not depend on whether this object can be initialized to a specific state or not.

7 Conclusion and open problems

In this paper, we proved that for all $n \geq 4$ there is no deterministic *and* linearizable $(n, 2)$-set agreement object that is equivalent to the $(n, 2)$-set agreement task, and this is because any such object can be used to solve 2-consensus. We conjecture that for all $n > k > 2$ there is no deterministic linearizable (n, k)-set agreement object that is equivalent to the (n, k)-set agreement task, and this is because any such object can be used to solve some (n', k')-set agreement task that is strictly stronger than the (n, k)-set agreement task according to the partial order of set agreement tasks shown by Chaudhuri and Reiners [6]. The techniques we used in this paper to prove Theorems 19 and 29, however, do not seem appropriate to approach this conjecture: As n and k increase, the number of sequential behaviours for the first k accesses of an (n, k)-DLSA object increases exponentially with k, and this would be overwhelming for $k \geq 3$.

We also proved that there is no *oblivious* deterministic and linearizable $(3, 2)$-set agreement object that is equivalent to the $(3, 2)$-set agreement task; the case of (not necessarily oblivious) $(3, 2)$-set agreement objects is still open.

References

1. Elizabeth Borowsky and Eli Gafni. *The implication of the Borowsky-Gafni simulation on the set-consensus hierarchy.* CSD (Series). UCLA Computer Science Department, 1993. URL: `https://books.google.ca/books?id=gpNeGwAACAAJ`.
2. Elizabeth Borowsky, Eli Gafni, and Yehuda Afek. Consensus Power Makes (Some) Sense! (Extended Abstract). In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '94, pages 363–372, New York, NY, USA, 1994. ACM. `doi:10.1145/197917.198126`.
3. Armando Castañeda, Michel Raynal, and Sergio Rajsbaum. Specifying Concurrent Problems: Beyond Linearizability. *CoRR*, abs/1507.00073, 2015. `arXiv:1507.00073`.
4. David Yu Cheng Chan, Vassos Hadzilacos, and Sam Toueg. On the Number of Objects with Distinct Power and the Linearizability of Set Agreement Objects. In Andréa W. Richa, editor, *31st International Symposium on Distributed Computing (DISC 2017)*, volume 91 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 12:1–12:14, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.DISC.2017.12`.
5. Soma Chaudhuri. More Choices Allow More Faults: Set Consensus Problems in Totally Asynchronous Systems. *Inf. Comput.*, 105(1):132–158, July 1993. `doi:10.1006/inco.1993.1043`.
6. Soma Chaudhuri and Paul Reiners. Understanding the Set Consensus Partial Order Using the Borowsky-Gafni Simulation (Extended Abstract). In *Proceedings of the 10th International Workshop on Distributed Algorithms*, WDAG '96, pages 362–379, London, UK, UK, 1996. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=645953.675630`.
7. Maurice Herlihy and Nir Shavit. The Topological Structure of Asynchronous Computability. *J. ACM*, 46(6):858–923, November 1999. `doi:10.1145/331524.331529`.
8. Maurice Herlihy and Jeannette Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990. `doi:10.1145/78969.78972`.
9. Gil Neiger. Set-Linearizability. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '94, pages 396–, New York, NY, USA, 1994. ACM. `doi:10.1145/197917.198176`.
10. Michael Saks and Fotios Zaharoglou. Wait-free K-set Agreement is Impossible: The Topology of Public Knowledge. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC '93, pages 101–110, New York, NY, USA, 1993. ACM. `doi:10.1145/167088.167122`.

A Characterization of Consensus Solvability for Closed Message Adversaries

Kyrill Winkler
TU Wien, Vienna, Austria
kwinkler@ecs.tuwien.ac.at

Ulrich Schmid
TU Wien, Vienna, Austria
https://ti.tuwien.ac.at/ecs/people/schmid
s@ecs.tuwien.ac.at

Yoram Moses
Technion, Haifa, Israel
moses@ee.technion.ac.il

Abstract

Distributed computations in a synchronous system prone to message loss can be modeled as a game between a (deterministic) distributed algorithm versus an omniscient message adversary. The latter determines, for each round, the directed communication graph that specifies which messages can reach their destination. Message adversary definitions range from oblivious ones, which pick the communication graphs arbitrarily from a given set of candidate graphs, to general message adversaries, which are specified by the set of sequences of communication graphs (called admissible communication patterns) that they may generate. This paper provides a complete characterization of consensus solvability for closed message adversaries, where every inadmissible communication pattern has a finite prefix that makes all (infinite) extensions of this prefix inadmissible. Whereas every oblivious message adversary is closed, there are also closed message adversaries that are not oblivious. We provide a tight non-topological, purely combinatorial characterization theorem, which reduces consensus solvability to a simple condition on prefixes of the communication patterns. Our result not only non-trivially generalizes the known combinatorial characterization of the consensus solvability for oblivious message adversaries by Coulouma, Godard, and Peters (Theor. Comput. Sci., 2015), but also provides the first combinatorial characterization for this important class of message adversaries that is formulated directly on the prefixes of the communication patterns.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Dynamic networks, Consensus, Message Adversary

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.17

Funding *Kyrill Winkler*: Austrian Science Fund (FWF): ADynNet (P28182) and RiSE/SHiNE (S11405)

1 Introduction

With the increasing pervasiveness of mobile wireless devices that are subject to energy constraints and unreliable communication, there is a growing need for a profound theoretical understanding of what can and what cannot be computed in such *dynamic networks* [15].

One popular way to model these systems is by assuming a set of n synchronous, infallible[1] agents (called processes in the sequel), which communicate by exchanging messages over directed links whose behavior is under the control of a *message adversary* (MA) [1]. Whereas

[1] Of course, the typical behavior exhibited by a crashed process (i.e., no messages sent after it failed) can easily be mimicked by an appropriate message adversary as well.

© Kyrill Winkler, Ulrich Schmid, and Yoram Moses;
licensed under Creative Commons License CC-BY
23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller; Article No. 17; pp. 17:1–17:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

the power of such a message adversary must be restricted somehow, as it may otherwise simply suppress all messages and make any distributed task trivially impossible, it is considered omniscient and may hence seek to actively foil the success of any given solution algorithm within its constraints. We stress that an algorithm that has been proved to solve some task, like consensus, under a certain message adversary MA provides a correct result, irrespective of whether erroneous communication stems from random faults or malicious interference, as long as it remains within the specification of the MA.

Even more fundamental than solution algorithms are characterizations of message adversaries with respect to certain tasks, that is, statements about when a message adversary becomes too powerful for solving a task: If a message adversary can be shown to cause any solution algorithm to fail under some admissible communication pattern, there is no hope to find a correct algorithm. In this paper, we provide a complete characterization of the important class of closed message adversaries (see below for its definition) with respect to the classic *deterministic consensus problem*, where each process starts with some input value and, eventually, every process has to decide the same value that was the input of some process. Thanks to the close relation between the solvability of consensus and variants of common knowledge in epistemic reasoning [13], our findings are also relevant for the latter (c.f. Section 5.2).

Classes of Message Adversaries

The fundamental objects in the message adversary model are *communication graphs* G_r, which, for a given round r, determine which messages are delivered and which are lost (see Section 2 for details). A *communication pattern* is a sequence $G_1, G_2, \ldots$ of such communication graphs for round $1, 2, \ldots$; a *message adversary* (MA) is just the set of its *admissible* communication patterns MA.

One important and well-studied class of message adversaries are *oblivious* ones [6], where the MA may pick every G_r arbitrarily, i.e., without restrictions, from a given set $\mathbf{D}$ of candidate graphs. We can hence write $\mathsf{MA} = \mathbf{D}^\omega$ for an oblivious message adversary. A prominent example is the message adversary that allows a certain number of mobile link failures per round, which was studied in the seminal work by Santoro and Widmayer [23]. Consensus has been shown to be impossible if $\mathbf{D}$ contains all communication graphs where $\leqslant n-1$ edges are missing (excluding self-loops).

Oblivious message adversaries do not allow to change the set of candidate graphs over time, however, which makes them unsuitable to model transient performance variations in real-world dynamic networks. In wireless mobile ad-hoc networks, for example, there may be substantial periods of time where some nodes are outside the communication range of others, e.g. in disaster-relief applications [18] or under strong interference by other radio transmitters [14]. Another source of time-varying communication conditions are mode switches, caused by the nodes themselves, due to reasons such as energy-saving, boot-up completion or fault recovery.

In this paper, we therefore focus on the class of *limit-closed* message adversaries (subsequently called *closed*[2] MAs for brevity), which are a proper superset of oblivious message adversaries. Their characterizing property is that every sequence $(\sigma_i)_{i\geq 1}$, such that each σ_i

[2] In the topological framework of [21], which follows-up on [2], a closed message adversary corresponds to a compact space, which is why, in a topological context, the term "compact message adversary" is preferable. Because we do not use topological reasoning in this paper, we stick to the terminology of [19] in this paper and use the term "closed message adversary" instead.

is the *prefix* of an admissible communication pattern and each σ_i is a prefix of σ_{i+1}, has a limit $\sigma \in \mathsf{MA}$. Equivalently, for every communication pattern σ that is inadmissible under a closed message adversary, there exists a round r such that all extensions of the round r prefix of σ are inadmissible as well. Note that this definition makes the set of admissible sequences of the MA closed and hence a safety property in the spirit of [2]. Thus, for closed message adversaries, reliable message delivery implies that the message delivery is reliable and bounded and fairness implies bounded fairness. Even though this is a very strict requirement, our characterization is more general than the previously existing combinatorial[3] characterizations of consensus solvability under message adversaries with an arbitrary number of processes [6, 23].

More specifically, it follows directly from its definition that an oblivious message adversary is closed, but there are also closed message adversaries that are not oblivious. An important example are MAs that ensure *bounded instability*. Such message adversaries guarantee some "global stabilization round" r_{GST}, which must be bounded (with some known bound), such that the communication graphs become "nice" from round r_{GST} onwards. The latter could occur in a myriad of ways, however: For example, starting from r_{GST}, a benign dynamic graph structure (like a vertex-stable source component [3] that persists for sufficiently many rounds) could appear. Alternatively, the number of rounds it takes for some processes to reach all other processes could be bounded, analogous to partially synchronous systems [9][4] or MAs corresponding to the ones from [16, 17]. A different example are MAs that assemble communication patterns by concatenating finite sequences of communication graphs, picked from a set of communication graph sequences with a fixed maximal length. Such a MA can either be allowed to choose an arbitrary combination of elements from this set, similar to oblivious message adversaries, or be subject to constraints, such as bounded instability.

On the other hand, relaxing the above requirement of r_{GST} being bounded to being unbounded but finite provides an illustrative example for a *general* message adversary. A MA that guarantees stability only eventually, i.e., after *unbounded*[5] *instability*, is not closed. In fact, the communication pattern where stability *never* occurs does not have a finite prefix such that all extensions are inadmissible, as any such finite prefix could still be made admissible by attaching a stability phase. Whereas we do not aim at a combinatorial characterization of consensus solvability for non-closed MAs in this paper (a topological characterization can be found in [21]), we believe that research on this challenging problem might benefit from our result.

An example for $n = 2$

In order to illustrate the different message adversary classes, we consider the case where the set of processes is $\Pi = \{\circ, \bullet\}$, i.e., $n = 2$, and the communication graphs are $\circ \leftarrow \bullet$, $\circ \leftrightarrow \bullet$, $\circ \rightarrow \bullet$, and $\circ \quad \bullet$ (in the last graph, there is no edge between the processes).

[3] We use the term "combinatorial" to distinguish classic approaches that essentially enumerate combinatorial objects (like execution prefixes) from the topological characterization provided in [21], which is more general but rests on fairly abstract topological concepts like connected subspaces of infinite executions. Among the advantages of combinatorial characterizations is that they are often directly amenable to algorithmic implementations and hence more operational.

[4] In contrast to [9], message adversaries allow us to restrict precisely which processes may communicate with each other and which may not.

[5] Note that finite but unbounded r_{GST} is equivalent in terms of task solvability to r_{GST} being bounded with an unknown bound: The code of a solution algorithm $\mathcal{A}$ for the latter cannot depend on any bound on r_{GST}, yet must work for every finite value of r_{GST}.

In this example, an oblivious message adversary is represented by a subset of these communication graphs and considers all sequences admissible that consist exclusively of graphs from this subset. Thus, every oblivious message adversary that permits the communication graph $\circ \quad \bullet$ makes solving consensus trivially impossible. Furthermore, the oblivious message adversary that permits the communication graphs $\circ\!\leftarrow\!\bullet$, $\circ\!\leftrightarrow\!\bullet$, and $\circ\!\rightarrow\!\bullet$ is known to make consensus impossible at least since [23]. However, removing only one graph from this set of possible communication graphs already makes consensus solvable: If the message adversary permits $\circ\!\leftarrow\!\bullet$ and $\circ\!\leftrightarrow\!\bullet$, both processes may decide the input of $\bullet$ after the first round. If it permits $\circ\!\leftarrow\!\bullet$ and $\circ\!\rightarrow\!\bullet$, a process may decide on the other's input if it received the other's message in the first round and on its own input otherwise.

We get an example of a closed message adversary by allowing the communication graphs $\circ\!\leftarrow\!\bullet$, $\circ\!\leftrightarrow\!\bullet$, and $\circ\!\rightarrow\!\bullet$, provided there is some known round r_{GST} by which it is guaranteed that the same communication graph has occurred consecutively, in rounds r and $r+1$. This message adversary is closed, because every limit of a sequence $(\sigma_i)_{i\geq 1}$ of admissible prefixes, s.t. σ_i is a prefix of σ_{i+1}, is admissible here: Intuitively, the reason is that every prefix in the sequence that is longer than r_{GST} rounds, in order to be admissible, must have rounds $r, r+1 \leq r_{GST}$ that satisfy the property described above, and every continuation of such a sequence, which corresponds to a limit sequence, is also admissible. We can hence use Algorithm 1 introduced in Section 4 for solving consensus.

The same message adversary, however with the property that r_{GST} is finite but unbounded, provides an example of a non-closed message adversary. Consensus is solvable even under this message adversary; a suitable algorithm has been provided in [25]. Still, our Algorithm 1 does not work anymore, since the MA is not closed: Since the repetition of the same graph twice in a row may occur arbitrarily late, there is a limit sequence $(\sigma_i)_{i\geq 1}$ that consists entirely of admissible prefixes where it never occurs. This is an inadmissible communication pattern for this message adversary, however.

Contributions and Paper Organization

In this paper, we provide a complete combinatorial characterization of consensus solvability under closed message adversaries. Compared to the topological characterization provided in [21] (see the related work below), it is considerably less abstract and, more importantly, also fully operational: it utilizes an easy to check property of (finitely many) *prefixes* of admissible communication patterns, rather than properties of (uncountably many) infinite communication patterns. Compared to the combinatorial characterization for oblivious message adversaries developed in [6], our characterization applies to the larger class of closed MAs, and relies on checking the simple dynamic graph property "non-empty kernel intersection" (see Theorem 1 below), rather than on an involved algorithm that exploits certain properties of the so-called "β-classes".

In more detail, we present a condition on the admissible communication patterns of a message adversary, which we prove to be both necessary and sufficient for solving consensus. The condition, given in Theorem 1 below, rests on two main ingredients:

(i) An equivalence relation $\sigma|_r \sim \rho|_r$ on the r-round prefixes of the communication patterns σ, ρ, which is the transitive closure of the per-process equivalence relations $\sigma|_r \sim_{p_i} \rho|_r$; the latter holds if process p_i cannot distinguish $\sigma|_r$ from $\rho|_r$, in every round up to r.

(ii) The set of processes (called the *kernel* of $\sigma|_r$, denoted by $\mathrm{Ker}(\sigma|_r)$) that influence every process in the system within $\sigma|_r$. The processes in $\mathrm{Ker}(\sigma|_r)$ are the ones that manage to broadcast their initial value to all processes within $\sigma|_r$, and are hence sometimes called *broadcasters*.

Whereas it is not too difficult to prove (see Theorem 2) that solving consensus within r rounds under a communication pattern σ requires the existence of a broadcaster in $\sigma|_r$, i.e., $\mathrm{Ker}(\sigma|_r) \neq \emptyset$, this is only a *necessary* condition. What needs to be added to also make it sufficient is that actually *all* transitively indistinguishable prefixes $\rho|_r$ must have some *common* element(s) in their kernels $\mathrm{Ker}(\rho|_r)$: With $[\sigma|_r]$ denoting the equivalence class w.r.t. $\sim$ containing $\sigma|_r$, which can be computed from the message adversary specification, our main result is the following:

▶ **Theorem 1.** *Consensus is solvable under a closed message adversary MA if and only if for each $\sigma \in MA$ there is a round r such that $\bigcap_{x \in [\sigma|_r]} Ker(x) \neq \emptyset$.*

Theorem 1 is not only interesting from a theoretical point of view, but may also have practical implications in that it allows to avoid attempts to develop consensus algorithms for dynamic networks that do not allow any solution. The remainder of this paper is devoted to the proof of Theorem 1 and is organized as follows: In Section 2, we present our system and computation model, the definition of our message adversaries and their properties, and the specification of the consensus problem. In Section 3, we use indistinguishability arguments and König's Infinity Lemma to prove that consensus is impossible if the condition in Theorem 1 does not hold. In Section 4, we prove the sufficiency of our condition, by specifying an algorithm that solves consensus under a message adversary that satisfies Theorem 1. A comparison to other approaches in Section 5 and some conclusions in Section 6 round-off our paper.

Related Work

Dynamic networks have been studied in a wide variety of different forms in the distributed computing literature, e.g. as predicates on "heard-of" sets [5], in the form of dynamic connectivity constraints (T-interval connectivity) [16, 17], as time-varying graphs [4], and many more (c.f. the overview in [15]). The term message adversary was introduced in [1], as an intuitive way to understand message loss in distributed computing systems. [22] investigated the relation between message adversaries and failure detectors, whereas [3, 25] studied eventually stabilizing message adversaries.

Perhaps one of the earliest characterizations of consensus solvability in synchronous distributed systems prone to communication errors is the seminal work by Santoro and Widmayer [23], where it was shown that consensus is impossible if up to $n-1$ messages may be lost in each round. This classic result was refined in [24] and, more recently, by Coulouma et al. in [6], where a property of an equivalence relation on the sets of communication graphs was found that captures exactly the source of consensus impossibility under an oblivious message adversary. The authors also showed how this property can be exploited in order to develop a generic consensus algorithm.

The first characterization of consensus solvability under general message adversaries was provided in [11], albeit only for systems that consist of two processes. A bivalence argument was used there to show that certain communication patterns, namely, a fair or a special pair of unfair communication patterns, must be excluded by the MA for consensus to become solvable. In [21], Nowak et al. provided a complete topological characterization for systems of arbitrary size, which relies on non-trivial extensions of the seminal work by Alpern and Schneider [2]. It focuses on the space of communication patterns (actually, the corresponding infinite sequences of process-time graphs resp. configurations), and defines topologies based on variants of the well-known common-prefix metric. The authors show that consensus is solvable for a given MA if and only if this space partitions into multiple components $PS(v)$

consisting of the executions with decision value v. For closed message adversaries, these sets are shown to be compact and hence closed, while for non-closed MAs they are only relatively compact. The existence of this partitioning is tied to the non-empty kernel intersection of every individual $PS(v)$ in both cases. For closed MAs, it has been shown that $PS(v)$ can be replaced by some approximation $PS^{\varepsilon}(v)$, which consist of the k-prefixes of the sequences in $PS(v)$ for $\varepsilon = 2^{-k}$. In a way, the condition given in Theorem 1 can hence be viewed as the combinatorial counterpart of this topological characterization.

Regarding the study of closed message adversaries, the seminal works by Dolev et al. [8] and Dwork et al. [9] on partially synchronous systems introduced important abstractions like eventual stabilization and eventually bounded message delays, and provided a characterization of consensus solvability under various combinations of synchrony and failure models (including byzantine-faulty processes).

In [19], Lubitch and Moran provided a general consensus impossibility result for asynchronous distributed systems. For this purpose, they studied the configuration tree of an asynchronous system, in which each node corresponds to a configuration and a node is the child of another, if the corresponding configuration is a successor configuration of the other. They focused on closed subsets of runs of asynchronous distributed algorithms, defined by the property that every path in the configuration tree corresponds to an admissible execution (which is equivalent to limit-closure). Using a model-independent construction of closed schedulers that generate closed sets of runs, they provided a unified impossibility proof for consensus using a bivalence argument. The simplicity of their approach rests on the fact that the forever bivalent run so constructed belongs to the closed set of runs, and is hence admissible. This convenient property of closed sets of communication patterns is also used in our paper, albeit we use König's Infinity Lemma rather than a bivalence argument for constructing a non-deciding run. We briefly explain at the end of Section 3 why our model is not suitable for a bivalence argument like the one in [19].

2 Model of Computation

We consider a finite set $\Pi = \{p_1, \ldots, p_n\}$ of deterministic network-coupled state machines (processes) with unique identifiers (for brevity we assume the identifier of p_i is its index i), unlimited memory and infinite computational power. Processes, which are assumed to be fault-free, operate in lock-step synchronous rounds where they can exchange messages with each other via unidirectional pairwise links. A message can only arrive at the destination in the same round in which it is sent. However, not all messages are guaranteed to reach their recipient. Instead, a message adversary controls which messages arrive and which get lost in a round. The message adversary is omniscient, yet restricted by a set of rules that are known to the processes. The processes need to cooperate to solve the distributed consensus problem (to be defined later), which the message adversary, in turn, seeks to foil. Throughout this paper, we study the conditions under which there is a winning strategy for either side.

Message Adversaries

A round r *communication graph* G_r is a directed graph in which each vertex corresponds to a process and there is an edge $(p_i \to p_j)$ in G_r if and only if a message sent by p_i to p_j in round r is not lost. We denote by $\mathrm{In}_{G_r}(p_i)$ the incoming edges of p_i in G_r. We assume that every process always receives a message from itself, hence G_r contains self-loops at all nodes. A finite sequence of consecutive communication graphs $G_r, G_{r+1}, \ldots, G_{r+k}$ is called a *finite communication pattern*. We say that a finite communication pattern of the

form $\sigma = G_r, \ldots, G_{r+k}$, has *length* $|\sigma| = k+1$ and *range* $[r, r+k]$. Infinite communication patterns $\sigma = G_1, G_2, G_3, \ldots$ are called communication patterns for brevity. If σ is an infinite or finite communication pattern of range at least $[1, r]$, then $\sigma|_r$ denotes the r-round *prefix* of σ, i.e., the first r graphs $G_1, G_2, \ldots, G_r$ of σ. A message adversary is a set of communication patterns MA, (the specification of) which is assumed to be common knowledge.

For a given communication pattern $\sigma = G_1, G_2, \ldots$, we define influence in the usual way (see e.g. [17]): we say process p_i at round r influences p_j in round r', written as $(p_i, r) \rightsquigarrow_\sigma (p_j, r')$, if there is a chain of messages, starting at p_i no earlier than the beginning of round $r+1$ and ending at p_j no later than at the end of r'. Formally, influence is just the transitive closure (over rounds) of the relation $(p_i, r) \rightarrow_\sigma (p_j, r+1)$, which holds if the edge $(p_i \rightarrow p_j) \in G_{r+1}$. The *view* of a process in round r for communication pattern σ is a graph $\mathsf{view}_{\sigma|_r}(p_i) = \langle V, E \rangle$, such that V is the collection of process-round pairs that have influenced p_i by round r, i.e., $V = \{(p_j, r') \colon (p_j, r') \rightsquigarrow_\sigma (p_i, r)\}$ and there is an edge in $E \subseteq V \times V$ precisely if a corresponding message was successfully delivered and subsequently the recipient of this message influenced p_i, i.e., $E = \{((p_j, r') \rightarrow (p_k, r'+1)) : (p_j \rightarrow p_k) \in G_{r'+1} \wedge (p_k, r'+1) \rightsquigarrow_\sigma (p_i, r)\}$.

We say that two finite communication patterns ρ, σ are indistinguishable for p_i, written as $\rho \sim_{p_i} \sigma$, if $\mathsf{view}_\rho(p_i) = \mathsf{view}_\sigma(p_i)$. We note that $\mathsf{view}_\rho(p_i) = \mathsf{view}_\sigma(p_i)$ implies that $|\rho| = |\sigma|$ because we assume that every communication graph contains self-loops. We use $\rho \sim \sigma$ to denote the transitive closure of the above relation, over all processes and sequences, i.e., w.r.t. a set of finite communication patterns S, we write $\rho \sim \sigma$ if, for some multiset of processes $\{p_{i_1}, \ldots, p_{i_k}\}$ of Π and some set $\{\tau_1, \ldots, \tau_{k-1}\} \subseteq S$, we have $\rho \sim_{p_{i_1}} \tau_1 \sim_{p_{i_2}} \ldots \sim_{p_{i_{k-1}}} \tau_{k-1} \sim_{p_{i_k}} \sigma$. We note that $\sim$ is an equivalence relation and, given the set S (usually, a set of prefixes of admissible communication patterns), we denote by $[\sigma]$ the equivalence class of σ over the set S. Given $S' \subseteq S$ with $\sigma \in S'$, the subclass $[\sigma]_{S'}$ of σ is the equivalence class of σ on S'; we will sloppily write $[\sigma]_{S'} \subseteq [\sigma]$ in this case. Note carefully that the transitive closure $\sim$ of $[\sigma]_{S'}$ runs over sequences in S' only. Hence, there may be $\rho \in S'$ with $\rho \in [\sigma]$ but $\rho \notin [\sigma]_{S'}$, namely, when every path between σ and ρ contains some sequence in $S \setminus S'$. Finally, we extend $\sim_p$ (resp. $\sim$) to infinite communication patterns σ, ρ, by writing $\sigma \sim_p \rho$ (resp. $\sigma \sim \rho$) if and only if $\sigma|_r \sim_p \rho|_r$ (resp. $\sigma|_r \sim \rho|_r$), for every finite $r \geq 1$.

We define the *kernel* of a prefix $\sigma|_r$ by $\mathrm{Ker}(\sigma|_r) = \{p_i \in \Pi \mid \forall p_j \in \Pi \colon (p_i, 0) \rightsquigarrow_\sigma (p_j, r)\}$ to be the set of those processes that reach every other process, directly or via transitive messages by the end of round r. The kernel intersection $\mathrm{KI}[\sigma|_r]$ of an equivalence class $[\sigma|_r]$ is defined as $\mathrm{KI}[\sigma|_r] = \bigcap_{x \in [\sigma|_r]} \mathrm{Ker}(x)$. For an infinite communication pattern σ, we define $\mathrm{Ker}(\sigma) = \bigcup_{r>0} \mathrm{Ker}(\sigma|_r)$.

Executions

Starting from its initial state $\mathsf{state}_0(p_i)$, the state of p_i at the end of its round r computation is denoted as $\mathsf{state}_r(p_i)$. The collection of all round r states is called a round r configuration C_r. The state transition function that guides the evolution of $\mathsf{state}_r(p_i)$ to $\mathsf{state}_{r+1}(p_i)$ (which also depends on $\mathrm{In}_{G_{r+1}}(p_i)$), as well as the message sending function that computes (from $\mathsf{state}_r(p_i)$) the message to be broadcast[6] in round $r+1$, are specified by an algorithm in pseudo-code. Note that if $(p_j, r) \rightsquigarrow_\sigma (p_i, s)$ then $\mathsf{state}_s(p_i)$ may depend on $\mathsf{state}_r(p_j)$ but not necessarily on $\mathsf{state}_{r+1}(p_j)$. An execution is an infinite sequence of configurations

[6] We assume that the same message is sent to every receiver for simplicity, which makes sense since the algorithm does not necessarily know even n.

$C_0, C_1, \ldots$ that is in accordance with the state transition and message sending function of some deterministic algorithm. Since the processes are deterministic, the initial configuration C_0, together with the communication pattern, uniquely determines the execution, hence we write $\langle C_0, \sigma \rangle$ for the run that results when executing a given algorithm that starts from C_0 and is subject to an infinite communication pattern σ.

We say that two executions $\varepsilon, \varepsilon'$ are indistinguishable to process p_i, written as $\varepsilon \sim_{p_i} \varepsilon'$, if p_i goes through the same sequence of states in both executions. We note that, from the previous arguments, we immediately have that two executions $\langle C_0, \sigma \rangle \sim_{p_i} \langle C_0', \sigma' \rangle$ are indistinguishable for p_i if the corresponding communication patterns $\sigma \sim_{p_i} \sigma'$ are indistinguishable for p_i and, for all p_j that influence p_i in σ, we have that $\mathsf{state}_0(p_j)$ is the same in C_0 and in C_0'. Intuitively, this holds because a process can only be certain about the (successful and unsuccessful) message exchanges and initial states that it has either observed directly (by itself) or indirectly (by being influenced accordingly).

Consensus

In the classic distributed consensus problem, each process $p_i \in \Pi$ holds an input value $x_i \in \mathbb{N}$ and an output or decision value y_i, initialized to $y_i = \perp$, that can be written to at most once. In an execution of a consensus algorithm, the initial configuration C_0 is usually just an assignment of input values to each process. We say that p_i decides v if p_i assigns v to y_i. An algorithm that correctly solves consensus ensures the following:

(Decision) Every process p_i decides on a value y_i eventually.

(Agreement) If $y_i \neq \perp$ and $y_j \neq \perp$, then $y_i = y_j$.

(Validity) If $y_i = v$ and $v \neq \perp$, then $v = x_j$ for some process p_j.

We commence by identifying a crucial relation between decision values and $\mathrm{Ker}(\sigma)$.

▶ **Theorem 2.** *Let MA be an arbitrary message adversary, let C_0 be an initial configuration, let $\sigma \in$ MA, and let $p_i \in \Pi$. The decision y_i in execution $\varepsilon = \langle C_0, \sigma \rangle$ of any correct consensus algorithm $\mathcal{A}$ satisfies $y_i = x_j$ for some $p_j \in$ Ker(σ).*

Proof. Suppose that there is an initial configuration C_0 and a $\sigma \in \mathsf{MA}$, such that, in some correct consensus algorithm $\mathcal{A}$, some process decides v in execution $\langle C_0, \sigma \rangle$ where $v \neq x_i$ for any $p_i \in \mathrm{Ker}(\sigma)$. By validity, the set $\overline{K} = \{p_i \in \Pi \setminus \mathrm{Ker}(\sigma) \mid x_i = v\}$ is non-empty. Assuming some arbitrary ordering on $\overline{K} = \{p_{i_1}, \ldots, p_{i_k}\}$, let $C_0^0 = C_0$ and for $0 < m \leqslant |\overline{K}| = k$, let C_0^m be the same as C_0^{m-1} except that $x_{i_m} \neq v$ in C_0^m. We show by induction that, for any $\ell \geq 0$, some process decides v in the computation of $\mathcal{A}$ with execution $\varepsilon_\ell = \langle C_0^\ell, \sigma \rangle$. But then, $\mathcal{A}$ violates validity, because in C_0^k, $x_i \neq v$ for every process $p_i \in \Pi$.

The induction base for $\ell = 0$ follows from the initial assumption. For the induction step from $\ell - 1 \geq 0$ to ℓ, we observe that since $\overline{K} \subseteq \Pi \setminus \mathrm{Ker}(\sigma)$, there is some process p_i such that, for any round r, $(p_{i_\ell}, 0) \not\rightsquigarrow_\sigma (p_i, r)$.

By construction, therefore, $\varepsilon_{\ell-1} \sim_{p_i} \varepsilon_\ell$. The induction hypothesis asserts that some process decides v in $\varepsilon_{\ell-1}$ and thus, by agreement, p_i decides v in $\varepsilon_{\ell-1}$ and hence in ε_ℓ as well. ◀

Because of Theorem 2, it makes sense to only consider message adversaries MA where for each $\sigma \in \mathsf{MA}$ we have $\mathrm{Ker}(\sigma) \neq \emptyset$. Perhaps not surprisingly, though, this requirement alone is insufficient for solving consensus. Investigating precisely what is additionally required is the goal of the remaining paper.

3 Necessity of an Eventually Common Kernel

In this section, we prove the "only if"-direction of Theorem 1, showing that consensus is impossible under a message adversary that contains a communication pattern σ with $\mathrm{KI}[\sigma|_r] = \bigcap_{x\in[\sigma|_r]} \mathrm{Ker}(x) = \emptyset$ for all rounds r.

Our general proof strategy, as realized in Lemma 7, is to show that, because of the above condition, there is a particular subclass $X \subseteq [\sigma]$ of communication patterns with $\bigcap_{x\in X} \mathrm{Ker}(x) = \emptyset$. In Lemma 8, we show that this makes consensus impossible because it would mean that there are two indistinguishable executions with a different decision value. To prove Lemma 7, we apply König's Infinity Lemma to a tree that can be described as follows: At level r, the nodes are the subclasses of $[\sigma|_r]$ that consist of prefixes ρ such that $\mathrm{Ker}(\rho)$ is a subset of an "associate kernel" in some fixed set S of kernels. We then apply König's Infinity Lemma to show that there exists an infinite path in the tree that corresponds to a non-empty subclass X of $[\sigma]$. Because the associate kernels satisfy $\bigcap_{K\in S} K = \emptyset$, this implies $\bigcap_{x\in X} \mathrm{Ker}(x) = \emptyset$, as required.

▶ **Lemma 3** (König's Infinity Lemma, cf. [7, Chapter 6]). *Let $V_1, V_2, \ldots$ be an infinite sequence of disjoint non-empty finite sets, and let G be a graph on their union. Assume that every vertex v in a set V_r with $r > 1$ has a neighbor $f(v)$ in V_{r-1}. Then G contains an infinite path $v_1 v_2 \ldots$ with $v_r \in V_r$ for all r.*

We start with the definition of *kernel-restricted* classes $[\sigma]^{\mathbf{K}}$ (resp. $[\sigma]^{\subseteq\mathbf{K}}$), where all members must have a kernel that is equal to (resp. a subset of) an element of the non-empty set of kernels $\mathbf{K} = \{K_1, \ldots, K_k\}$:

▶ **Definition 4** (Kernel-restricted classes). *Given the equivalence class $[\sigma]_S$ of a communication pattern σ over the set S, and some non-empty set of non-empty kernels $\emptyset \neq \mathbf{K} = \{K_1, \ldots, K_k\}$, the kernel-restricted class $[\sigma]^{\mathbf{K}} \subseteq [\sigma]_S$ of σ is defined to be the class $[\sigma]_{S'}$ of σ over the set $S' = \{\rho \in S : Ker(\rho) \in \mathbf{K}\}$. Similarly, $[\sigma]^{\subseteq\mathbf{K}} \subseteq [\sigma]_S$ is the class $[\sigma]_{S''}$ over the set $S'' = \{\rho \in S : \exists K_i \in \mathbf{K}$ s.t. $Ker(\rho) \subseteq K_i\}$.*

Note that $[\sigma]^{\mathbf{K}} = \emptyset$ if $\mathrm{Ker}(\sigma) \notin \mathbf{K}$ (although this does not happen in Lemmas 5 to 7 below), and that there may be $\rho \in [\sigma]$ with $\mathrm{Ker}(\rho) \in \mathbf{K}$ but $\rho \notin [\sigma]^{\mathbf{K}}$, which happens if every path connecting $\sigma \sim \rho$ in $[\sigma]$ contains at least one prefix not in S'.

▶ **Lemma 5.** *Let MA be a message adversary that contains some σ s.t. $KI[\sigma|_r] = \emptyset$ holds for all rounds r. Then, there is a non-empty set $\mathbf{K} = \{K_1, \ldots, K_k\}$ of kernels, such that $\bigcap_{i=1}^{k} K_i = \emptyset$ and $[\sigma|_r] = [\sigma|_r]^{\mathbf{K}}$ for infinitely many rounds $r = r_1, r_2, \ldots$.*

Proof. For an arbitrary round $r > 0$, let $g([\sigma|_r]) = \{\kappa \subseteq \Pi \colon \exists \rho \in [\sigma|_r]$ with $\mathrm{Ker}(\rho) = \kappa\}$. Since Π is a finite set, the power set $\mathcal{P}(\Pi)$ is a finite set as well. By the pigeonhole principle, there is some set $\mathbf{K} = \{K_1, \ldots, K_k\} \subseteq \mathcal{P}(\Pi)$ such that, for infinitely many rounds r, $g([\sigma|_r]) = \mathbf{K}$, hence $[\sigma|_r] = [\sigma|_r]^{\mathbf{K}}$. Note that, since obviously $\sigma|_r \in [\sigma|_r]$ for every $r > 0$, we have $\mathrm{Ker}(\sigma|_r) \in \mathbf{K}$. By the assumption that, for all r, $\mathrm{KI}[\sigma|_r] = \emptyset$, we also have $\bigcap_{i=1}^{k} K_i = \emptyset$. ◀

▶ **Lemma 6.** *Let MA be a message adversary. If there exist a $\sigma \in MA$ and a set $\mathbf{K} = \{K_1, \ldots, K_k\} \subseteq \mathcal{P}(\Pi)$ with $[\sigma|_r] = [\sigma|_r]^{\mathbf{K}}$ for infinitely many rounds r, then there is an infinite sequence $\mathcal{V} = V_1, V_2, \ldots$ of sets $V_i \subseteq [\sigma|_i]^{\subseteq\mathbf{K}}$, such that, for all $i \geq 1$, each of the following holds:*

(1) $V_i \neq \emptyset$

(2) $V_i = \{\nu_1, \ldots, \nu_{m(i)}\}$ *for a finite $m(i) > 0$ s.t., for $1 \leqslant j \leqslant m(i)$: $\sigma|_i \in \nu_j \subseteq [\sigma|_i]^{\subseteq\mathbf{K}}$*

(3) *Each $\nu \in V_{i+1}$ has a neighbor $\nu' = f_{i+1}(\nu) \in V_i$*

Proof. Initializing $V_i = \emptyset$ for every $i \geq 1$, we construct V_i, starting from $i = 1$, as follows: For each of the infinitely many indices $i = r$ where $[\sigma|_r] = [\sigma|_r]^{\mathbf{K}}$, we set $V_i = \{[\sigma|_i]^{\mathbf{K}}\}$; note that $\mathrm{Ker}(\sigma|_r) \in \mathbf{K}$ in this case, so (1) and (2) hold for V_i. Moreover, for all $1 \leqslant j < i$, we add to V_j the set $\{\rho|_j : \rho \in [\sigma|_i]^{\mathbf{K}}\}$. As $\mathrm{Ker}(\rho|_j) \subseteq \mathrm{Ker}(\rho)$, and since $\rho \sim \tau$ implies also $\rho|_j \sim \tau|_j$, (1) and (2) continue to hold for V_j. Moreover, for $\nu \in V_{i+1}$, we define $f(\nu) = \nu|_i$, which secures (3). Thus, the infinite sequence of sets $\mathcal{V}$ with properties (1)–(3) exists, as claimed. ◀

▶ **Lemma 7.** *Let MA be a message adversary that contains some $\sigma \in$ MA such that, for all r, we have $KI[\sigma|_r] = \emptyset$. Then there is a nonempty set of kernels $\mathbf{K} = \{K_1, \ldots, K_k\}$ with $\bigcap_{i=1}^{k} K_i = \emptyset$ and a non-empty kernel-restricted subclass $X \subseteq [\sigma]^{\subseteq \mathbf{K}}$ with $\bigcap_{x \in X} Ker(x) = \emptyset$.*

Proof. We take σ and apply to it Lemma 5 and then Lemma 6. In this manner, we obtain the nonempty set of kernels $\mathbf{K}$ and an infinite tree spanning the members of the infinite sequence of sets $V_1, V_2, \ldots$ via the neighbor functions $f_i : V_i \to V_{i-1}$. König's Infinity Lemma ensures that there is an infinite path in this tree, which yields $X = \lim_{i \to \infty} V_i$: For every $\rho \in X$, we have $\rho|_r \sim \sigma|_r$ for every $r \geq 1$, and hence $\rho \sim \sigma$. ◀

▶ **Lemma 8.** *Consensus is impossible under a message adversary MA with $\sigma \in$ MA such that some non-empty subclass $X \subseteq [\sigma]$ satisfies $\bigcap_{x \in X} Ker(x) = \emptyset$.*

Proof. We show that the existence of some algorithm $\mathcal{A}$ that solves consensus under MA would lead to a contradiction. Since $X \neq \emptyset$, for some $k > 0$, there is a multiset of processes $\{p_{i_1}, \ldots, p_{i_{k-1}}\}$, which may be empty (if $k = 1$), and a non-empty multiset of communication patterns $Y = \{\sigma_1, \ldots, \sigma_k\} \subseteq X$ with $\sigma_j \in [\sigma]$ for $1 \leqslant j \leqslant k$ and $\bigcap_{\rho \in Y} \mathrm{Ker}(\rho) = \emptyset$, such that $\sigma_1 \sim_{p_{i_1}} \sigma_2 \ldots \sigma_{k-1} \sim_{p_{i_{k-1}}} \sigma_k$.

Let C_0^v be the input assignment where $x_i = v$ for all processes p_i and let $C_0^{\overline{v}}$ be the input assignment where $x_i = \overline{v}$ for all processes p_i, for some $v \neq \overline{v}$. By validity, in execution $\langle C_0^v, \sigma_1 \rangle$ every process running $\mathcal{A}$ decides v, while in $\langle C_0^{\overline{v}}, \sigma_1 \rangle$ they decide $\overline{v}$. Since input assignments are not restricted in any way, toggling the input values of $p_1, p_2, \ldots$ from v to $\overline{v}$, one after the other, reveals that there are input assignments C_0', C_0'' that differ only in the input value of a single process p_i, yet every process running $\mathcal{A}$ decides v in $\varepsilon_1^v = \langle C_0', \sigma_1 \rangle$ and $\overline{v}$ in $\varepsilon_1^{\overline{v}} = \langle C_0'', \sigma_1 \rangle$.

A simple induction shows that v is decided in $\varepsilon_j^v = \langle C_0', \sigma_j \rangle$ and $\overline{v}$ is decided in $\varepsilon_j^{\overline{v}} = \langle C_0'', \sigma_j \rangle$ for $1 \leqslant j \leqslant k$. The base case, $\ell = 1$, was already shown above. For the step from ℓ to $\ell + 1$ with $1 \leqslant \ell < k$, we have by hypothesis that v was decided in ε_ℓ^v and $\overline{v}$ was decided in $\varepsilon_\ell^{\overline{v}}$. Since $\sigma_\ell \sim_{p_{i_\ell}} \sigma_{\ell+1}$, we have $\varepsilon_\ell^v \sim_{p_{i_\ell}} \varepsilon_{\ell+1}^v$ and v is also decided in $\varepsilon_{\ell+1}^v$. A similar argument shows that $\overline{v}$ is decided in $\varepsilon_{\ell+1}^{\overline{v}}$.

We conclude the proof by showing that $p_i \in \mathrm{Ker}(\sigma_j)$ for $1 \leqslant j \leqslant k$, and hence $p_i \in \mathrm{Ker}(\rho)$ for any $\rho \in Y$, which contradicts $\bigcap_{\rho \in Y} \mathrm{Ker}(\rho) = \emptyset$. Suppose, for some j, $p_i \notin \mathrm{Ker}(\sigma_j)$. Hence there is some p_k with $p_i \not\rightsquigarrow_{\sigma^j} p_k$. Since C_0' and C_0'' are the same except for the input of p_i, $\varepsilon_j^v \sim_{p_k} \varepsilon_j^{\overline{v}}$, and p_k decides the same in ε_j^v and in $\varepsilon_j^{\overline{v}}$. This, however, contradicts our previous statement that v is decided in ε_j^v and $\overline{v}$ is decided in $\varepsilon_j^{\overline{v}}$ for some $v \neq \overline{v}$. ◀

A note on bivalence arguments

At this point, the reader might wonder why we did not resort to a bivalence argument, as introduced in [12] and used heavily in the literature (e.g. in the very closely related papers [6, 11, 19, 20, 23, 24]) to establish our impossibility result. In a nutshell, in the case of binary consensus, bivalence proofs establish impossibility by inductively constructing a

run where every reached configuration is bivalent. Reachable configurations are classified according to whether only 0-decided (resp. 1-decided) configurations are reachable from it, in which case the configuration is called 0-valent (resp. 1-valent), or whether both a 0-decided and a 1-decided configuration are reachable from it, in which case the configuration is called bivalent. Note carefully that the agreement property implies that no process can have decided in a bivalent configuration, as a single decision, say, to 0, would make the configuration already 0-valent.

In the bivalence induction proof, it is first established that not all initial configurations can be 0-valent or 1-valent. Then, under the hypothesis that the reached round r configuration is bivalent, it is shown that not all round $r+1$ configurations can be univalent. This results in a forever bivalent run, in which no process can have decided. Care must be taken, however, to also prove that the run so constructed is also admissible, as the processes must decide only in an admissible run.

The reason why we cannot use such an argument in our setting, and need to resort to König's Lemma instead, is that the induction step might lead to a dead end later on: there is no a priori guarantee that the bivalent successor chosen in some step is one that allows the construction of an infinite admissible suffix. Technically, what would be needed in the induction step to ensure this is that two configurations that are currently indistinguishable for some process can be extended in a way that remains indistinguishable forever for this process. As the only thing we know about our message adversary is that it is closed and eventually guarantees a non-empty kernel intersection, however, it is not clear how to infer sufficient information on the possible communication graphs generated in all admissible suffixes to guarantee this.

4 Sufficiency of an Eventually Common Kernel

Lemma 8 established the "only if"-direction of Theorem 1. We now show the "if"-direction of Theorem 1, by introducing Algorithm 1. This algorithm solves consensus under any message adversary MA that guarantees, for every $\sigma \in \mathsf{MA}$, that there is a round r and a non-empty set $K \subseteq \Pi$ such that $K = \mathrm{KI}[\sigma|_r] = \bigcap_{x \in [\sigma|_r]} \mathrm{Ker}(x)$. Note that this algorithm could stop operating immediately after decision, as the decision happens in the same round at all processes.

Essentially, each process p_i executing the algorithm attempts to send a local estimation of the communication pattern, stored in array E, along with its own input value $x[i]$ to all other processes. Here, the kth entry of E, $\mathsf{E}[k]$, contains the local estimate of the round k communication graph. On reception of a round r message from some process p_j, including $p_j = p_i$, p_i stores the input of p_j in $x[j]$ and adds the edge $(p_j \to p_i)$ to $\mathsf{E}[r]$ since it could only have received the message if $(p_j \to p_i) \in G_r$. Note that this implies that in every round r, the edge $(p_i \to p_i)$ is added to $\mathsf{E}[r]$, as we assumed that every process receives a message from itself in each round. Process p_i then proceeds to merge its local estimates E with the ones received and then calculates the set S of all communication patterns $G_1, \ldots, G_r$ that it considers possible. It does this by checking which communication pattern prefixes, allowed by MA, are in accordance with what p_i observed so far. Finally, p_i picks an arbitrary prefix ρ of S and checks whether all members of the equivalence class of ρ have a non-empty intersection K of their kernels. If this is the case, p_i decides on the input of the process in K with the largest identifier. We note that the equivalence class of ρ can be computed from the specification of MA, which is, according to the system model, known to the processes. Note that there are only finitely many communication graphs and hence finitely many round r prefixes at any finite r.

Algorithm 1 Consensus algorithm, code for process p_i.

Initialization:
1 $\mathsf{x}[i] \leftarrow x_i$
2 $\mathsf{x}[j] \leftarrow \bot$ for $j \neq i$
3 $\mathsf{E}[0] \leftarrow \emptyset$
4 $r \leftarrow 1$

Transmit round r messages:
5 Attempt to send $\langle \mathsf{E}, \mathsf{x} \rangle$ to all
6 Receive $\langle \mathsf{E}_j, \mathsf{x}_j \rangle$ from all p_j with $(p_j \to p_i) \in G_r$

Round r computation:
7 **foreach** p_j *from which* p_i *received a message in round* r **do**
8 | **foreach** k *with* $\mathsf{x}_j[k] \neq \bot$ **do**
9 | | $\mathsf{x}[k] \leftarrow \mathsf{x}_j[k]$
10 | Add $(p_j \to p_i)$ to $\mathsf{E}[r]$
11 | **if** $r > 1$ **then**
12 | | **for** $r'' \in \{1, \dots r-1\}$ **do**
13 | | | $\mathsf{E}[r''] \leftarrow \mathsf{E}[r''] \cup \mathsf{E}_j[r'']$
14 **for** $r' \in \{1, \dots r\}$ **do**
15 | $\mathsf{V}_{r'} \leftarrow \{p_j \in \Pi \colon \exists p_k \in \Pi \text{ s.t. } (p_k \to p_j) \in \mathsf{E}[r']\}$
16 Let $\mathrm{In}_{G_{r'}}(\mathsf{V}_{r'})$ denote the edges $(u \to v) \in G_{r'}$ with $v \in \mathsf{V}_{r'}$
17 $\mathsf{S} \leftarrow \left\{\sigma|_r = G_1, \dots, G_r \colon \sigma \in \mathsf{MA} \text{ and } \mathrm{In}_{G_{r'}}(\mathsf{V}_{r'}) = \mathsf{E}[r'] \text{ for all } 1 \leqslant r' \leqslant r\right\}$
18 Pick an arbitrary $\rho \in \mathsf{S}$
19 **if** $y_i = \bot$ *and there exists* $K \neq \emptyset$ *s.t.* $K = KI[\rho]$ **then**
20 | $m \leftarrow \max\{j : p_j \in K\}$
21 | $y_i \leftarrow \mathsf{x}[m]$ /* decide */
22 $r \leftarrow r + 1$

In the following proof of the correctness of Algorithm 1, we use $\mathsf{var}_i^r \in \mathsf{state}_r(p_i)$ to denote the value of variable var, held by process p_i at the end of its round r computation. This is clearly the same as the value of var after it was written to the last time in round r, so if the last write to var occurs in line ℓ, var_i^r is the value of var at process p_i after p_i finished line ℓ for the last time in round r.

We start with a few technical results, which essentially assert the correctness of the local estimates. Lemma 9 establishes that the set E approximates edges in the communication graph faithfully if there was an appropriate influence.

▶ **Lemma 9.** $(p_j, r') \rightsquigarrow_\sigma (p_i, r) \Leftrightarrow \mathrm{In}_{G_{r'}}(p_j) \subseteq \mathsf{E}_i^r[r']$.

Proof. For the "$\Rightarrow$" direction, we show inductively for $r' < \ell \leqslant r$ that $(p_j, r') \rightsquigarrow_\sigma (p_k, \ell)$ implies $\mathrm{In}_{G_{r'}}(p_j) \subseteq \mathsf{E}_k^\ell[r']$.

For $\ell = r' + 1$, $(p_j, r') \rightsquigarrow_\sigma (p_k, \ell)$ implies, by definition of the $\rightsquigarrow_\sigma$ relation, $(p_j \to p_k) \in G_{r'+1}$, i.e., p_k receives the round $r' + 1$ message of p_j. By Line 10, $\mathrm{In}_{G_{r'}}(p_j) \subseteq \mathsf{E}_j^{r'}[r']$. Since p_k received $\mathsf{E}_j^{r'}[r']$ from p_j via its round $r' + 1$ message, p_k incorporates $\mathsf{E}_j^{r'}[r']$ into its own $\mathsf{E}_k^{r'+1}[r']$ in Line 13.

For $r' + 1 < \ell \leqslant r$, we assume that $(p_j, r') \rightsquigarrow_\sigma (p_k, \ell - 1)$ implies $\mathrm{In}_{G_{r'}}(p_j) \subseteq \mathsf{E}_k^{\ell-1}[r']$. If for some p_m, $(p_j, r') \rightsquigarrow_\sigma (p_m, \ell)$, by definition, $(p_k \to p_m) \in G_\ell$ for some p_k with $(p_j, r') \rightsquigarrow_\sigma (p_k, \ell - 1)$. By hypothesis, $\mathrm{In}_{G_{r'}}(p_j) \subseteq \mathsf{E}_k^{\ell-1}[r']$, hence p_m receives $\mathsf{E}_k^{\ell-1}[r']$ at the beginning of round ℓ and incorporates $\mathrm{In}_{G_{r'}}(p_j)$ into $\mathsf{E}_m^\ell[r']$ in Line 13 of its round ℓ computation.

The "$\Leftarrow$" direction holds trivially if $p_i = p_j$. If $p_i \neq p_j$, then, according to Line 13, p_i can only learn about $\mathrm{In}_{G_{r'}}(p_j)$ if there is a chain of messages, starting at p_j no earlier than the end of round r' and ending at p_i before its round r computing step. By definition, hence $(p_j, r') \rightsquigarrow_\sigma (p_i, r)$. ◀

Lemma 10 shows that any $\mathsf{E}_k[r']$ is an under-approximation of $G_{r'}$, i.e., it does not contain any fabricated edges.

▶ **Lemma 10.** *If $(p_j \to p_i) \in \mathsf{E}_k^r[r']$ then (1) $(p_j \to p_i) \in G_{r'}$ and (2) $(p_i, r') \rightsquigarrow_\sigma (p_k, r)$.*

Proof. (1) Suppose $(p_j \to p_i) \in \mathsf{E}_k^r[r']$ but $(p_j \to p_i) \notin G_{r'}$. Since $(p_j \to p_i)$ could only be added to $\mathsf{E}_k^r[r']$ through Line 10 or Line 13, we have $(p_j \to p_i) \in \mathsf{E}_i^{r'}[r']$. Since Line 10 is guarded by Line 7, $(p_j \to p_i) \in \mathsf{E}_i^{r'}[r']$ can only happen when p_i received a message from p_j in round r', which implies that $(p_j \to p_i) \in G_{r'}$.

(2) If $(p_j \to p_i) \in \mathsf{E}_k^r[r']$, since $(p_j \to p_i) \in \mathrm{In}_{G_{r'}}(p_i)$, because during the loop of Line 7, every in-edge is added in Line 10, $\mathrm{In}_{G_{r'}}(p_i) \subseteq \mathsf{E}_k^{r'}[r']$. By Corollary 9 hence $(p_i, r') \rightsquigarrow_\sigma (p_k, r)$. ◀

The above lemmas can be combined to the following Corollary 11, which shows that if $\mathsf{E}_k^r[r']$ contains some non-empty fraction of the incoming round r' edges of some process p_i, then it actually contains all the incoming round r' edges of p_i and vice-versa. Furthermore, this is equivalent to the existence of an influence from p_i in round r' to p_k in round r.

▶ **Corollary 11.** *The following are all equivalent: (1) $(p_j \to p_i) \in \mathsf{E}_k^r[r']$, (2) $\mathrm{In}_{G_{r'}}(p_j) \subseteq \mathsf{E}_k^r[r']$, (3) $(p_i, r') \rightsquigarrow_\sigma (p_k, r)$.*

Corollary 11 can be used to show that S_i^r indeed contains all execution prefixes that are indistinguishable from the current execution prefix, as expressed in Corollary 12.

▶ **Corollary 12.** *In every round r of an execution $\langle C_0, \sigma\rangle$, the following holds: $\mathsf{S}_i^r = \{\rho|_r : \rho \in \mathsf{MA}$ and $\rho|_r \sim_{p_i} \sigma|_r\}$.*

Proof. Since we assume self-loops in every communication graph, by Corollary 11, for any processes p_i, p_j, p_k, an in-edge $(p_k \to p_j)$ to p_j is present in $\mathsf{E}_i^r[r']$ exactly when $(p_j, r') \rightsquigarrow_\sigma (p_i, r)$. Hence, S_i^r contains exactly those sequences $\rho_r = G_1, \ldots, G_r$ where each G_ℓ matches the in-neighborhood for those nodes p_m that satisfy $(p_m, \ell) \rightsquigarrow_\sigma (p_i, r)$. By definition, these are precisely those prefixes $\rho|_r$ of $\rho \in \mathsf{MA}$ where $\mathsf{view}_{\rho|_r}(p_i) = \mathsf{view}_{\sigma|_r}(p_i)$ and thus $\rho|_r \sim_{p_i} \sigma|_r$. ◀

Finally, from the assumption that the specification of MA is known to the processes, and since the finite number of processes implies a finite number of communication graphs, which, in turn, implies a finite number of round r prefixes, we have the following Corollary 13 of Lemmas 9 and 10.

▶ **Corollary 13.** *Let r be an arbitrary round. The equivalence class $[\sigma|_r]$ is finite and thus can be computed by all processes.*

We are now ready to show the correctness of Algorithm 1.

▶ **Lemma 14.** *Algorithm 1 solves consensus under every message adversary MA that ensures, for all $\sigma \in \mathsf{MA}$, that $KI[\sigma|_r] \neq \emptyset$ for some round r.*

Proof. Pick an arbitrary sequence $\sigma \in \mathsf{MA}$ and fix some input assignment C_0. We show that Algorithm 1 satisfies all properties of the consensus specification in the execution $\langle C_0, \sigma \rangle$. Let t be the earliest round such that $\mathrm{KI}[\sigma|_t] \neq \emptyset$, i.e., for all $t' < t$ we have $\mathrm{KI}[\sigma|_{t'}] = \emptyset$. We show that (1) no p_i decides before round t and (2) there is a value v, which is the input value of some process p_j, such that every undecided p_i decides v in round t.

(1) Suppose some process p_i decides in some round $t' < t$. This means p_i passes the guard of Line 19 in round t'. After executing Line 18, due to Corollary 12, at any process p_i, $\rho_i^{t'}$ is set to some prefix $\sigma'|_{t'}$ with $\sigma' \in \mathsf{MA}$ and $\sigma'|_{t'} \sim_{p_i} \sigma|_{t'}$. By definition, hence $\sigma'|_{t'} \in [\sigma|_{t'}]$ and, in fact, $[\sigma'|_{t'}] = [\sigma|_{t'}]$. This, taken together with Corollary 13, yields that, after performing the computations in the guard of Line 19, at every process p_i, we have $K_i^{t'} = \mathrm{KI}[\sigma|_{t'}]$. If p_i passes this guard, we have $K_i^{t'} \neq \emptyset$ which contradicts the assumption that t is the earliest round for which the condition holds.

(2) By assumption, we have some set K such that $\mathrm{KI}[\sigma|_t] = K \neq \emptyset$. Let m be the maximal identifier of any process in K. We show the claim for $v = x_m$. Since we assumed that p_i has not decided yet, a similar argument as above shows that every process p_i passes the guard of Line 19 in round t and decides on $\mathsf{x}^t[m]$ via Line 21. Since $p_m \in \mathrm{Ker}(\sigma|_t)$, $(p_m, 0) \in \mathsf{view}^t_{\sigma|_t}(p_i)$ and hence Lines 1 and 9 ensure that $\mathsf{x}^t[m] = x_m = v$. ◀

5 Relation to Other Approaches

In this section, we will complement our results by exploring some not so obvious relations to other approaches. Moreover, we will provide some illustrating examples.

5.1 A topological view

Our combinatorial characterization of consensus for closed message adversaries given in Theorem 1 is of course compatible with the topological one established in [21], as mentioned already in our discussion of related work in Section 1: The requirement of broadcastability of the connected components in the topological space of admissible executions made in [21] is clearly enforced by our non-empty kernel intersection condition.

5.2 A knowledge-based view

Our results are also perfectly in line some well-known results of the epistemic analysis of consensus [13]. First and foremost, it is well-known (see e.g. [13, Prop. 4]) that simultaneous consensus can only be achieved if there is common knowledge of certain facts (in particular, a decision taking place). And indeed, Lemma 14 reveals that in Algorithm 1 all processes are guaranteed to decide in the same round. Furthermore, it was established in [10] that for simultaneous consensus, common knowledge of the input values is needed. We will therefore argue now that the processes actually establish common knowledge about the initial values of the members of the kernel intersection guaranteed by Theorem 1, in the same round.

The first thing to mention is that our similarity relation $\sim$ is actually equivalent to G-reachability in view-based interpretations in S5 models, cf. [13, p. 561]. For every σ and r, the equivalence class $[\sigma|_r]_\sim$, and hence also the kernel intersection $K_r = \bigcap_{x \in [\sigma|_r]_\sim} \mathrm{Ker}(x)$, can be computed by every process from the a priori common knowledge of MA. Since every process can unambiguously determine the equivalence class $[\sigma|_r]_\sim$ (albeit not $\sigma|_r$ itself) from its local state, all the algorithm has to do is to wait for the round where $K_r \neq \emptyset$ for the first time, which is the round r guaranteed by Theorem 1.

What is also worth exploring is the apparent paradox that our algorithm manages to attain some common knowledge in systems with unreliable communication that is not already available initially. According to [13, Thm. 5], this should be impossible. There is no contradiction here, however, since if we translate the definition of "communication cannot be guaranteed" [13, p. 566] into a corresponding message adversary MA, it would fail the condition of Theorem 1: Condition NG2 says that if some p_i does not receive a message in round r in $\sigma|_r$, then there is some $\rho|_r \in [\sigma|_r]_\sim$ where no message is delivered in round r. NG2 thus allows the non-empty kernel intersection condition of Theorem 1 to hold only if a simultaneous broadcast happens in round r in all prefixes in $[\sigma|_r]_\sim$. This, in turn, violates condition NG1, which requires that every $\sigma|_{r-1}$ can be continued with a suffix where no further messages are ever received. Consequently, closed message adversaries that satisfy our condition do not qualify as "communication cannot be guaranteed".

6 Conclusions

We have derived the, to the best of our knowledge, first combinatorial characterization of consensus solvability in the important class of closed message adversaries: Consensus is solvable here if and only if, for every communication pattern, eventually, the kernel-intersection of all transitively indistinguishable communication pattern prefixes becomes non-empty. Our consensus characterization surpasses all existing non-topological characterizations for message adversaries known so far w.r.t. the range of message adversaries covered. Moreover, unlike the existing topological characterization for closed message adversaries, it operates on prefixes of communication patterns and is hence well-suited for practical implementations as well. Moreover, our result was obtained using only very basic notions from dynamic networks, like the kernel of a communication pattern prefix and the transitive closure of the indistinguishability relation.

Regarding future work, a natural question is whether there is also a combinatorial characterization of consensus solvability for general (non-closed) message adversaries. Existing algorithms show that a simultaneous decision is sometimes impossible under such message adversaries, which suggests that our closed algorithm is definitely not applicable there. On the other hand, the principles exploited in Theorem 1 rest fundamentally only on the consensus specification itself, thus it seems unlikely that they could completely disappear even in a more general result.

References

1 Yehuda Afek and Eli Gafni. Asynchrony from Synchrony. In Davide Frey, Michel Raynal, Saswati Sarkar, RudrapatnaK. Shyamasundar, and Prasun Sinha, editors, *Distributed Computing and Networking*, volume 7730 of *Lecture Notes in Computer Science*, pages 225–239. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-35668-1_16.

2 Bowen Alpern and Fred B. Schneider. Defining Liveness. *Information Processing Letters*, 21(4):181–185, 1985.

3 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.

4 Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. Time-varying graphs and dynamic networks. *IJPEDS*, 27(5):387–408, 2012.

5 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.

6 Étienne Coulouma, Emmanuel Godard, and Joseph G. Peters. A characterization of oblivious message adversaries for which Consensus is solvable. *Theoretical Computer Science*, 584:80–90, 2015. doi:10.1016/j.tcs.2015.01.024.

7 Reinhard Diestel. *Graph Theory (3rd ed.)*. Springer, 2006.

8 Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

9 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the Presence of Partial Synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

10 Cynthia Dwork and Yoram Moses. Knowledge and common knowledge in a Byzantine environment: Crash failures. *Information and Computation*, 88(2):156–186, 1990. doi:10.1016/0890-5401(90)90014-9.

11 Tristan Fevat and Emmanuel Godard. Minimal Obstructions for the Coordinated Attack Problem and Beyond. In *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011 - Conference Proceedings*, pages 1001–1011, 2011. doi:10.1109/IPDPS.2011.96.

12 Michael J. Fischer, Nancy A. Lynch, and M. S. Paterson. Impossibility of Distributed Consensus with one Faulty Process. *Journal of the ACM*, 32(2):374–382, April 1985.

13 Joseph Y. Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. *J. ACM*, 37(3):549–587, 1990. doi:10.1145/79147.79161.

14 Wolfgang Kiess and Martin Mauve. A Survey on Real-world Implementations of Mobile Ad-hoc Networks. *Ad Hoc Networks*, 5(3):324–339, April 2007. doi:10.1016/j.adhoc.2005.12.003.

15 F. Kuhn and R. Oshman. Dynamic networks: Models and algorithms. *SIGACT News*, 42(1):82–96, 2011.

16 Fabian Kuhn, Nancy A. Lynch, and Rotem Oshman. Distributed computation in dynamic networks. In *STOC*, pages 513–522, 2010.

17 Fabian Kuhn, Rotem Oshman, and Yoram Moses. Coordinated consensus in dynamic networks. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '11. ACM, 2011.

18 Franck Legendre, Theus Hossmann, Felix Sutton, and Bernhard Plattner. 30 Years of Wireless Ad Hoc Networking Research: What about Humanitarian and Disaster Relief Solutions? What are we still missing? In *International Conference on Wireless Technologies for Humanitarian Relief (ACWR 11)*, Amrita, India, 2011. IEEE.

19 Ronit Lubitch and Shlomo Moran. Closed Schedulers: A Novel Technique for Analyzing Asynchronous Protocols. *Distributed Computing*, 8(4):203–210, June 1995. doi:10.1007/BF02242738.

20 Yoram Moses and Sergio Rajsbaum. A Layered Analysis of Consensus. *SIAM J. Comput.*, 31(4):989–1021, 2002.

21 Thomas Nowak, Ulrich Schmid, and Kyrill Winkler. Topological Characterization of Consensus Under General Message Adversaries. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, pages 218–227, New York, NY, USA, 2019. ACM. doi:10.1145/3293611.3331624.

22 Michel Raynal and Julien Stainer. Synchrony weakened by message adversaries vs asynchrony restricted by failure detectors. In *Proceedings ACM Symposium on Principles of Distributed Computing*, PODC'13, pages 166–175, 2013.

23 Nicola Santoro and Peter Widmayer. Time is not a healer. In *Proceedings of the 6th Annual Symposium on Theoretical Aspects of Computer Science (STACS'89)*, LNCS 349, pages 304–313, Paderborn, Germany, February 1989. Springer-Verlag.

24 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.

25 Kyrill Winkler, Manfred Schwarz, and Ulrich Schmid. Consensus in rooted dynamic networks with short-lived stability. *Distributed Computing*, 32(5):443–458, October 2019. doi:10.1007/s00446-019-00348-0.

An Efficient Universal Construction for Large Objects

Panagiota Fatourou
Institute of Computer Science, Foundation for Research and Technology-Hellas,
Department of Computer Science, University of Crete, Heraklion, Greece
faturu@csd.uoc.gr

Nikolaos D. Kallimanis
Institute of Computer Science, Foundation for Research and Technology-Hellas, Heraklion, Greece
nkallima@ics.forth.gr

Eleni Kanellou
Institute of Computer Science, Foundation for Research and Technology-Hellas, Heraklion, Greece
kanelou@ics.forth.gr

Abstract

This paper presents L-UC, a universal construction that efficiently implements dynamic objects of large state in a wait-free manner. The step complexity of L-UC is $O(n + kw)$, where n is the number of processes, k is the interval contention (i.e., the maximum number of active processes during the execution interval of an operation), and w is the worst-case time complexity to perform an operation on the sequential implementation of the simulated object. L-UC efficiently implements objects whose size can change dynamically. It improves upon previous universal constructions either by efficiently handling objects whose state is large and can change dynamically, or by achieving better step complexity.

2012 ACM Subject Classification Computing methodologies → Concurrent computing methodologies; Theory of computation → Concurrent algorithms; Computing methodologies → Concurrent algorithms; Computing methodologies → Shared memory algorithms; Theory of computation → Shared memory algorithms

Keywords and phrases universal construction, concurrent object, shared memory, simulation, wait-freedom, large object

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.18

Funding This work was supported by the General Secretariat of Research and Technology in Greece through project Sentitour at Scale (T1EDK-02857).

1 Introduction

1.1 Motivation and Contribution

Multi-core processors are nowadays found in all computing devices. Concurrent data structures are frequently used as the means through which processes communicate in multi-core contexts, thus it is important to have efficient and fault-tolerant implementations of them. A *universal construction* [11, 12] provides an automatic mechanism to get a concurrent implementation of any data structure (or object) from its sequential implementation.

In this paper, we present L-UC, an efficient, wait-free universal construction that deals with dynamic objects whose state is large. *Wait-freedom* [11] ensures that every process finishes the execution of each operation it initiates within a finite number of steps. The step complexity of L-UC is $O(n + kw)$, where n is the number of processes in the system, k is the *interval contention*, i.e., the maximum number of processes that are active during the execution interval of an operation, and w is the worst-case time complexity to perform

© Panagiota Fatourou, Nikolaos D. Kallimanis, and Eleni Kanellou;
licensed under Creative Commons License CC-BY
23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller; Article No. 18; pp. 18:1–18:15
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

an operation on the sequential data structure. The step complexity of an algorithm is the maximum number of *shared memory accesses* performed by a thread for applying any operation on the simulated object in any execution.

A large number of the previously-presented universal constructions [1,2,5,7,8,11,12] work by copying the entire state of the simulated object locally, making the required updates on the local copy, and then trying to make the local copy shared by changing one (or a few) shared pointers to point to it. Copying the state of the object locally is however very inefficient when coping with large objects. L-UC avoids copying the entire state of the simulated object locally; in contrast, it applies the required changes directly on the shared state of the object. For doing so, processes need to synchronize when applying the changes. Previous universal constructions that apply changes directly to the shared data structure (e.g., [5]) synchronize on the basis of each operation. However, this results in high synchronization cost. To reduce this cost, L-UC applies a wait-free analog of the combining technique [8,9]: each process simulates, in addition to its own operation, the operations of other active processes. So, in L-UC, processes have to pay the synchronization cost once for a batch of operations and not for each distinct operation.

Sim [8,10] is a wait-free universal construction that implements the combining technique. In Sim, each process p that wants to apply an operation, first announces it in an *Announce* array. Then, p reads all other announced operations, makes a local copy of the shared state, applies all the operations it is aware of on this copy, and tries to update a shared variable to point to this local copy. P-Sim, the practical version of Sim (presented also in [8]) is highly efficient for objects whose state is small. L-UC borrows some of the ideas presented in [8]. Specifically, as P-Sim, L-UC uses an *Announce* array in which processes announce their operations, and employs bit vectors to figure out which processes have active operations at each point in time. However, the bit vector mechanism of L-UC is more elaborated than that of P-Sim, because the active processes have to agree on the set of operations that must be applied on the shared data structure before they attempt to perform any changes. In contrast to Sim, L-UC avoids copying locally the object's state. This makes L-UC appropriate for simulating large objects.

L-UC also borrows some ideas from the universal construction presented in [5] that copes with large objects. As in the universal construction in [5], in L-UC, each process uses a directory to store copies of the shared variables (e.g., the shared nodes) it accesses while executing operations on the data structure. L-UC combines this idea with the idea of implementing a wait-free analog of the combining technique. This way, L-UC achieves step complexity that is $O(n + kw)$. In scenarios of low contention, this bound can be much smaller than the $O(nw)$ achieved by the universal construction in [5]. Moreover, the universal construction in [5] have processes synchronize on the basis of every single operation, whereas in L-UC, processes synchronize once to execute a whole batch of operations.

1.2 Related Work

In [11], Herlihy studied how shared objects can be simulated, in a wait-free manner, using read-write registers and consensus objects. In the proposed universal construction, the simulated object is represented by a list of records. Each record stores information about an operation *op* (its type, its arguments, and its response) that has been performed on the simulated object. It also stores the state of the simulated object after all operations inserted in the list up until *op* (including it) have been applied on the implemented object in the order that they have been inserted in the list. To agree on which record will be inserted in the list next, each record additionally stores an n-consensus object. To ensure wait-freedom,

the algorithm also employs an announce array of n elements, where the n threads running in the system announce their operations, and stores a (strictly increasing) sequence number in each record, which illustrates the order in which this record was inserted in the list. Threads help the record of a thread i to be inserted as the j-th record in the list when $i = j \texttt{ mod } n$. The step complexity of the algorithm is $O(n^2)$. The space overhead of the algorithm is $O(n^3)$ and each register contains the entire state of the object and a sequence number growing infinitely large. Herlihy revisited wait-free simulation of objects in [12], where it presented a universal construction which uses `LL/SC` and `CAS` objects and achieves step complexity $O(n + s)$, where s is the total size of the simulated object. These algorithms [11, 12] are inappropriate for large objects, as they work by copying the entire state of the object locally.

Afek, Dauber and Touitou presented in [1] a universal construction that employs a tree structure to monitor which processes are *active*, i.e. which processes are performing an operation on the simulated object at a given time. This tree technique was combined with some of the techniques proposed in [11, 12] in order to get a universal construction for simulating large objects, which has step complexity $O(kw \log w)$.

Anderson and Moir presented in [3] a wait-free universal construction for simulating large objects. In their algorithms, a contiguous array is used to represent the state of the object. Specifically, the object state is stored in B data blocks of size S each. To restrict memory overhead, the algorithms operate under the following assumptions: each operation can modify at most T blocks and each thread can help at most $M \geq 2T$ other threads. The step complexity of the universal construction in [3] is $O((n/\min\{k, M/T\})\ (B + MS + nw))$.

In [7], Fatourou and Kallimanis presented the family of RedBlue adaptive universal constructions. The F-RedBlue algorithm achieves $O(min\{k, logn\})$ step complexity and uses $O(n^2 + s)$ `LL/SC` registers. However, F-RedBlue uses large registers and it is not able to simulate objects whose state is stored in more than one register. S-RedBlue uses small registers, but the application of an operation requires to copy the entire state of the simulated object and thus it is inefficient for large objects. LS-RedBlue and BLS-RedBlue improve the step complexity of the algorithms presented by Anderson and Moir in [3] for large objects.

In [6], Felber et al. present CX, a wait-free universal construction, suitable for simulating large objects. This universal construction keeps up to $2n$ instances of the object state. In order to perform an update on the shared object, a process first appends its request in a shared request queue and then attempts to obtain the lock of some of the object instances. We remark that each such object instance stores a pointer to a queue node. Subsequently, the process uses this pointer to produce a valid copy of the object by performing all operations that were contained in the shared queue starting from the pointed node. Notice that CX has space complexity $O(ns)$, where n is the number of processes and s is the total size of the simulated object.

1.3 Roadmap

The rest of this paper is organized as follows. Our model is discussed in Section 2. L-UC is presented in Section 3. Section 3.1 provides an overview of the way the algorithm works and its pseudocode. Section 3.2 presents a detailed description of L-UC. A discussion of its complexity is provided in Section 3.3 and a sketch of proof for its correctness in Section 3.4.

2 Model

We consider an *asynchronous* system of n *processes*, $p_1, \ldots, p_n$, each of which may fail by *crashing*. Threads communicate by accessing (shared) base objects. Each *base object* stores a value and supports some primitives in order to access its state. An *`LL/SC`* object supports the

atomic primitives LL and SC. LL(O) returns the value that is stored into O. The execution of SC(O, v) by a thread p_i, $1 \leq i \leq n$, must follow the execution of LL(O) by p, and changes the contents of O to v if O has not changed since the execution of p's latest LL on O. If SC(O, v) changes the value of O to v, *true* is returned and we say that the SC is successful; otherwise, the value of O does not change, *false* is returned and we say that the SC is not successful or it is failed.L-UC is presented using LL/SC objects (as is the case for Sim [8, 10]). However, in a practical version of it, L-UC will be implemented using CAS objects (as is the case for P-Sim [8, 10]). A CAS object O supports in addition to Read(O), the primitive CAS(O, u, v) which stores v to O if the current value of O is equal to u and returns *true*; otherwise the contents of O remain unchanged and *false* is returned.

A *universal* construction can be used to implement any shared object. A universal construction supports the ApplyOp(*req*, i) operation, which applies the operation (or *request*) *req* to the simulated object and returns the return value of *req* to the calling thread p_i. In this paper, the concepts of an operation and a request have the same meaning and are used interchangeably. A *universal construction* provides a routine, for each process, to implement ApplyOp.

An object O is *linearizable*, if in every execution α, it is possible to assign to each completed operation *op* (and to some of the uncompleted operations), a point $*_{op}$, called the *linearization point* of *op*, such that: $*_{op}$ follows the invocation and precedes the response of *op*, and the response returned by *op* is the same as the response *op* would return if all operations in α were executed sequentially in the order imposed by the linearization points.

A *configuration* is a vector that contains the values of the base objects and the states of the processes, and describes the system at some point in time. At the *initial configuration*, processes are in their initial state and the base objects contain initial values. A *step* is taken by some process whenever the process executes a primitive on a shared register; the step may also include some local computation that is performed before the execution of the primitive. An *execution* is a sequence of steps. The *interval contention* of an instance of some operation in an execution is the number of processes that are active during the execution of this instance. The *step complexity* of an operation is the maximum number of steps that any thread performs during the execution of any instance of the operation in any execution. *Wait-freedom* guarantees that every process finishes each operation it executes in a finite number of steps.

3 The L-UC Algorithm

This section presents L-UC, our wait-free universal construction for large objects.

3.1 Overview

The pseudocode for L-UC is provided in Listings 1 and 2. The state of the simulated data structure in L-UC is shared and it can be updated directly by any process. Each process p that wants to apply a request, first announces it in an *Announce* array. In addition to the *Announce* array, L-UC uses a bit vector *Toggles* of n bits, one for each process. A process p_i toggles its bit, *Toggles*[i], after announcing a new request. The use of *Toggles* implements a fast mechanism for informing other processes of those processes that have pending requests.

Each execution of L-UC can be partitioned into phases. In each phase $i \geq 1$, the set of requests that will be executed in the next phase is agreed upon by the processes that are active. Moreover, those requests that have been agreed upon in the previous phase are indeed executed.

Listing 1 Data structures used in L-UC and pseudocode for LSimApplyOp.

```
struct NewVar {     // node of list of newly allocated variables
    ItemSV *var;    // points to the ItemSV struct of the variable
    NewVar *next;   // points to the next element of the list
};

struct NewList {
    ItemSV *first;
};

struct State {
   boolean applied[1..n];
   boolean papplied[1..n];
   int seq;
   NewList *var_list;
   RetVal RVals[1..n]; // return values
};

struct DirectoryNode {
   Name name;       // variable name
   ItemSV *sv;      // data item for the variable
   Value val;       // value of the data item
};

struct ItemSV {  // data item for a variable
   Value val[0..1];// old and new values of data item
   int toggle;      // toggle shows the current value of data item
   int seq;
};

// Toggles is implemented as an integer of n bits; if n is big, more than one
    such integers can be used
shared Integer Toggles = < 0,...,0 >;
shared State S = < F,...,F >,< F,...,F >,0,< ⊥ >,< ⊥,...,⊥ >>;
shared OpType Announce[1..n] = {⊥, ..., ⊥};

// Private local variable for process p_i
Integer toggle_i = 2^i;

RetVal ApplyOp(request req){  // Pseudocode for process p_i
    Announce[i] = req;        // Announce request req
    toggle_i = −toggle_i;
    Add(Toggles, toggle_i);       // toggle p_i's bit by adding 2^i to Toggles
    Attempt();       // call Attempt twice to ensure that req will be performed
    Attempt();
    return S.rvals[i];        // p_i finds its return value into S.rvals[i]
}
```

A process p_i that wants to execute a new request, it first announces it in *Announce*, and then it toggles its bit in *Toggles*. Afterwards, it calls a function, called `Attempt`, twice: After the execution of the first instance of `Attempt` by p_i, it is ensured that the set of requests agreed upon in one of the phases that overlap the execution of the `Attempt`, contains p_i's request. After the execution of the second instance of `Attempt` by p_i, it is ensured that p_i's request has been applied.

L-UC uses an `LL/SC` object S which stores appropriate fields to ensure the required synchronization between the processes in each phase. The first phase (phase 1) starts at the initial configuration and ends when the first successful `SC` is applied on S. Phase $i > 1$ starts when phase $i - 1$ finishes and ends when the i-th successful `SC` is applied on S.

To decide which set of requests will be executed in each phase, S contains two bit vectors, called *applied* and *papplied*, of n bits each (one for each process). The current request initiated by a process p_i has not yet been applied, if $S.applied[i] \neq S.papplied[i]$. When this condition holds, we call the current request of process p_i *pending*.

Listing 2 Pseudocode for L-UC.

```
1  void Attempt(Request req) {                    // pseudocode for process p_i
2    ProcessIndex q, j;
3    State ls, tmp;
4    Set lact;
5    Directory D;
6    NewVar *pvar = new NewVar(), *ltop;
7    ItemSV sv, *psv = new ItemSV();

9    psv→⟨val, toggle, seq⟩ = <<⊥,⊥>,0,0>;
10   pvar→⟨var, next⟩ = <psv, null>;
11   for j=1 to 2 do {
12      D = ∅;                                      // initialize direcory D
13      ls = LL(S);                                 // create a local copy of S
14      lact = Toggles;                             // read active set
15      ltop = ls.var_list→first; // read pointer to the list of newly-allocated
                                        variables
16      tmp.seq = ls.seq + 1;
17      tmp.papplied[1..n] = ls.applied[1..n];
18      tmp.applied[1..n] = lact[1..n];          // p_i will later attempt to update S
                                        with tmp, so it sets the fields of tmp
                                        appropriately
19      tmp.rvals[1...n] = ls.rvals[1..n];
20      for q=1 to n do {
21         if (ls.applied[q] ≠ ls.papplied[q]) { // q's request is pending
22            foreach access of a variable x while applying request Announce[q]{
23               if (x is a newly allocated variable) {
24                  if(CAS(ltop→next, null, pvar)){
25                     psv = new ItemSV();
26                     psv→⟨val, toggle, seq ⟩ = <<⊥,⊥>,0,0>;
27                     pvar = new NewVar();
28                     pvar→⟨var, next⟩ = <psv, null>;
29                  }
                    // use node pointed by ltop→next as the new variable's metadata
30                  ltop = ltop→next;
31                  add <x, ltop→var, ltop→var.val[0]> to D;
32               } else {                       // x is not a newly allocated variable
33                  let svp be a pointer to the ItemSV struct for x;
34                  if (this access is a read instruction) {
                       // perform the request on the local copy of x (if any)
35                     if (x exists in D) read x from D;
36                     else {
37                        sv = LL(*svp);
38                        if (tmp.seq==sv.seq) add <x,svp,sv.val[1-sv.toggle]> to D;
39                        else if(tmp.seq>sv.seq) add <x,svp,sv.val[sv.toggle]> to D;
40                        else goto Line 48; // values read from S by p_i obsolete, so
                                                    start from scratch
41                     }
42                  } else if (the access is a write instruction) update x in D;
43               }
44            }
45            store into tmp.rvals[q] the return value;
46         }
47      }
48      if (!VL(S)) continue; // value read in S by p_i is obsolete, so start from
                                        scratch
49      foreach record <x, svp, v> in D {
50         if(svp→seq > tmp.seq) break;  // all requests have been applied, so leave
                                        the loop
51         else if(svp→seq == tmp.seq) continue; // the variable has been modified,
                                        so continue
52         else if(svp→toggle == 0) SC(*svp, <<svp→val[0],v>, 1, tmp.seq>);
53         else SC(*svp, <<v, svp→val[1]>, 0, tmp.seq>); // make update visible
54      }
55      tmp.var_list = new List();  tmp.var_list→first = null;
56      SC(S, tmp);                          // try to modify S
57   }
58 }
```

In each instance of `Attempt`, p_i copies the value of S in a local variable ls (line 13), records necessary changes that it makes to its fields in another local variable tmp (lines 16-19, 45, 55), and uses `SC` in an effort to update S to the value contained in tmp (line 56). Specifically,

p_i reads S on line 13 (by performing an LL) and $Toggles$ on line 14. It then copies $S.applied$ into $tmp.papplied$ (line 17) and $Toggles$ into $tmp.applied$ (line 18). Recall that the $applied$ and $papplied$ fields of S encode the requests that are to be performed in each phase. So, if the SC that p_i performs on line 56 succeeds, all processes that will read the value this SC will write to S, will attempt to perform the requests encoded by p_i in those fields.

Next, for each j, $1 \leq j \neq n$, p_i checks whether $ls.applied[j] \neq ls.papplied[j]$ (lines 20-21), and if this is so, it applies the request recorded in $Announce[j]$. To execute the pending requests recorded in S, a process p_i uses a caching mechanism as in [4,5]: When a process first accesses a shared variable (e.g., a variable of the simulated shared data structure), it maintains a copy of it in a directory, D (which is local to p_i). For each pending request recorded in S, the required updates are first performed by p_i in the local copies of the data items that are residing in the directory (lines 22-45). Read requests executed by p_i are also served using D. Only after it has finished the simulation of all pending requests, p_i applies the changes listed in the elements of its directory to the shared data structure (lines 49-53).

For each data item x of the simulated object's state, L-UC maintains a record (struct) of type $ItemSV$. This struct stores the old and the current value of the data item in an array val of two elements, a toggle bit that identifies the position in the val array from where the current value for x should be read, and a sequence number that is used for synchronization.

Note that S contains also a field seq that is incremented every time a successful SC on S is performed. This field identifies the current phase of the execution. Before performing an update on the shared data structure (lines 49-53), p_i validates the values of the seq field read in S ($tmp.seq$) and that stored in $ItemSV$ for x ($svp \rightarrow seq$). Only if $svp \rightarrow seq < tmp.seq$ (line 53), the update is performed since otherwise it is already obsolete, i.e., $S.seq$ is already greater than $tmp.seq$ and therefore the SC of line 56 by p_i will fail.

Both the old and the current values of x must be stored in $ItemSV$ in order to avoid the following bad scenario. Consider two processes p_i and p_j that simulate the same request req. Assume that p_i is ready to execute line 37 for some variable x, whereas p_j has finished the simulation of req (lines 49-53) and has started updating the shared data structure. Then, it might happen that p_i reads the updated version for x although it should have read the old version. For this reason, p_j stores the old value (in addition to the new value) in one of the entries of the val array and appropriately updates the toggle bit to indicate which of the two values is the new one. If p_i discovers that it is too slow (line 38), it reads the old value for x stored in the $1 - toggle$ entry of its val array. Notice that, to ensure wait-freedom, p_i should continue executing req (to cope with the case that p_j fails before performing all the required updates to the shared data structure).

When a new data item x is allocated while executing a set of requests, additional synchronization between the processes that execute this set of requests is required to avoid situations where several processes allocate, each, a different record for x. We use a technique similar to that presented in [5] to ensure that all these processes use the same allocated ItemSV structure for x. Specifically, L-UC stores into S a pointer (called var_list) to a list of newly created data items shared by all processes that read this instance of S. Each time a process p_i needs to allocate the j-th, $j \geq 1$, such data item, it tries to add a structure of type $NewVar$ as the j-th element of the list (line 24). If it does not succeed, some other process has already done so, so p uses this structure (by moving pointer $ltop$ to this element on line 15, and by inserting $ltop \rightarrow var$ in its dictionary on line 31).

We remark that the fields of $ItemSV$ must be updated in an atomic way using SC. This requires that registers in the system store two words which is impractical. However, we can utilize single-word registers by using indirection. Indirection can also be used to implement S using single-word registers.

3.2 Detailed Description of `Attempt`

In the following, we detail the implementation of function `Attempt`, presented in Algorithm 2. When `Attempt` is executed by some process p_i, p_i executes two iterations (line 11) of checking whether there are pending requests and of attempting to apply them, as follows. It initializes its local directory D (line 12), creates in ls a local copy of the state of the simulated object (line 13), and reads in *lact* the value of *Toggles* (line 14), thus obtaining a view of which processes have pending requests at the current point in time (i.e., calculating the set of pending requests). Furthermore, it locally stores into *ltop* a pointer to the current variable list of the simulated object (line 15). Recall that the state of the object is copied into local variable ls using an `LL` primitive. In case this instance of `Attempt` is successful in applying the pending requests, it will update the shared state of the system using an `SC` primitive. For this purpose, the local variable *tmp* is prepared in lines 16 to 18, to serve as the value that will be stored into the shared state in case of success.

After having read the state of the simulated object, as well as the state of the requests of the other processes, p_i can detect which requests are pending. For this purpose, it iterates through the (locally stored) state of each process (line 20) and checks whether the values of *papplied* and *applied* differ for this process (line 21). If so, the request of this process was still pending when `Attempt` read the value of *Toggles* and therefore, `Attempt` intents to apply it. Notice that the iteration through the *papplied* and *applied* values consist of local steps. Notice also that at most k out of n processes have active requests, meaning what the request application contributes to step complexity depends on k rather than n.

We remark that the request of a process is expressed as a piece of sequential code. Therefore, in order to apply the request of some process, an instance of `Attempt` has to run through the sequential code of this request and carry out the variable accesses that this request entails, i.e. `Attempt` has to apply the modifications that this request incurs on the simulated object's variables (line 22). We distinguish three cases, namely the case where an access creates a new variable, the case where an access reads a variable, and the case where an access modifies an already existing variable.

In the first case (line 24), the new variable, which was created and stored in local variable *pvar*, must be added to the shared list of variables of the simulated object. Recall that a pointer to the top of the variable list has been read by p_i and stored in local variable *ltop*. Recall also that all processes are trying to perform the announced requests in the same order. As with each instance of `Attempt`, so also the p_i instance of `Attempt` tries to add *pvar* to the top of the list using a `CAS` primitive (line 24). In case this is successful, the metadata of this variable is initialized. In case the `CAS` is unsuccessful, then some other process has updated the object's variable list since this instance of `Attempt` read it into *ltop*. Given that all processes follow the same order when trying to insert newly-allocated variables, the failure means that the variable has already been inserted in the shared list of variables of the simulated object. In either case, i.e. either successful or unsuccessful insertion by p_i, *ltop* is updated to point to the data item of the newly allocated variable. Furthermore, the newly added variable is included into the local variable dictionary (line 31).

In the second case (line 34), the access to be performed is a read to a variable of the simulated object. If `Attempt` already has a local copy of this variable in its dictionary, it reads the value from there. If no local copy is present in the dictionary (line 37), then `Attempt` reads the variable using an `LL` primitive (line 37). At the same time, it checks the sequence number of the value that it has read, and in case this sequence number is less or equal to the local sequence number stored in *tmp*, then `Attempt` considers that it is reading a valid value. This value is then added to the local dictionary. However, in case the variable's

sequence number is larger than the local sequence number, this hints that this instance of `Attempt` has been rendered obsolete by some other process that has already applied all requests that this instance of `Attempt` is applying. In order to find out if this is the case, `Attempt` verifies whether the state of S has changed since it last read it (line 48) and if so, it gives up the current iteration of the for loop of line 11.

Finally, in the third case (line 42), where the access is a write to an already existing variable. In case that the accessed variable already exists in the local dictionary, the update on the local dictionary (line 42), updates the variable's value stored in the local dictionary. Otherwise, the update (line 42) creates a new entry and stores the value of the variable. Once the sequential code for the current request has all been run through and all variable accesses for the request have been performed, the request returns a return value, which is stored by `Attempt` for the process to access (line 45).

Recall that any update to a variable of the simulated object is performed locally by `Attempt`. Therefore, once all active requests have been applied, `Attempt` has to write back the local updates to the shared variables of the simulated object (lines 49 - 53). Notice that once again, the sequence numbers of the local and shared copies are instrumental in detecting whether a variable has already been updated or not (lines 51 - 53). More specifically, the condition of line 51 checks if another process has already updated or not the value of the shared variable while trying to apply the same set of operations calculated in lines 17 - 19. In case that a process is very slow and the whole set of operations calculated in lines 17 - 19 is applied, the condition of line 52 fails, and the process breaks the execution (line 50) of the *for-loop* of lines 49 - 53. Finally, once the updates have been performed, `Attempt` tries to update S, before performing any remaining iteration of the for loop of line 11.

3.3 Step Complexity

By inspection of the pseudocode of ApplyOp, it becomes apparent that its step complexity is determined by the step complexity of `Attempt`. In a practical version of L-UC where S is implemented using indirection, lines 13 and 14 contribute $O(n)$ to performance, since the size of the data records that are read is $O(n)$. The body of the if statement of line 21 (i.e., lines 22-42) is executed $O(k)$ times, each time contributing a factor of $O(w)$ (because of the foreach statement of line 22). Note that searching an element in the dictionary, adding an element to it or removing an element from it does not cause any shared memory accesses, i.e., it causes only local computation. So, the cost of executing lines 23-45 is $O(1)$. Note also that at most $O(kw)$ elements are contained in each dictionary. Therefore, the *foreach* of line 49 contributes $O(kw)$ to the total cost. The rest of the code lines access only local variables and thus they do not contribute to the step complexity of the algorithm. We conclude that the step complexity of ApplyOp is $O(n + kw)$.

3.4 Sketch of Correctness Proof

This section provides a sketch of the correctness proof of L-UC.

We start with some useful notation. Let α be any execution of L-UC and assume that some thread p_i, $i \in \{1, ..., n\}$, executes $m_i > 0$ requests in α. Let req_j^i be the argument of the j-th call of L-UC by p_i and let π_j^i be the j-th instance of `Attempt` executed by p_i (Figure 1). Let C_0 be the initial configuration. Define as Q_j^i the configuration after the execution of the `Add` instruction of line 41; let $Q_0^i = C_0$. We use $Toggles[i]$, $i \in \{1, \ldots, n\}$, to denote the i-th bit of $Toggles$, and let $toggle_j^i$ be the value of p_i's local variable $toggle_i$ at the end of req_j^i.

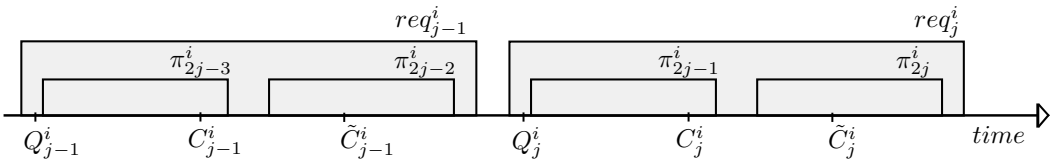


Figure 1 An example of an execution of L-UC.

In the following lemma, we argue that during the execution of each of the two iterations of the for loop of line 11 of any instance of `Attempt`, at least one successful `SC` instruction is performed.

▶ **Lemma 1.** *Consider any j, $0 < j \leq m_i$. There are at least two successful `SC` instructions in the execution interval of π^i_j.*

We continue with two technical lemmas. The first argues that the value of p_i's bit in the $Toggles$ array is equal to $j \mod 2$ after the execution of the j-th `Add` instruction of line 41 by p_i. It also shows that no process other than p_i can change this bit.

▶ **Lemma 2.** *For each j, $0 \leq j \leq m_i$, it holds that (1) $Toggles[i] = j \mod 2$ at Q^i_j, and (2) $Toggles[i]$ has the same value between Q^i_{j-1} and Q^i_j.*

The next lemma studies the value of $S.applied[i]$ after the execution of the j-th instance of `Attempt` by p_i.

▶ **Lemma 3.** *Consider any execution π^i_j, $j > 0$, of function `Attempt` by some thread p_i. $S.applied[i]$ is equal to $v = \lceil j/2 \rceil \mod 2$ just after the end of π^i_j.*

For each $l > 0$, let C_l be the configuration resulting after the execution of the l-th `Add` instruction in α. At C_0, $S.applied[i]$ is equal to *false*. Lemma 3 implies that just after π^i_1, $S.applied[i]$ is equal to *true*. Let C^i_1 be the first configuration between C_0 and the end of π^i_1 at which $S.applied[i]$ is equal to *true*. Consider any request req^i_j, $j > 1$. Lemma 3 implies that just after π^i_{2j-2}, $S.applied[i]$ is equal to $\lceil (j-2)/2 \rceil \mod 2 = (j-1) \mod 2$, while just after π^i_{2j-1}, $S.applied[i]$ is equal to $\lceil (2j-1)/2 \rceil \mod 2 = j \mod 2 \neq (j-1) \mod 2$. Let C^i_j be the first configuration between the end of π^i_{2j-2} and the end of π^i_{2j-1} such that $S.applied[i]$ is equal to $j \mod 2$. Figure 1 illustrates the above notation.

Since the value of $S.applied[i]$ can change only by the execution of `SC` instructions on S, it follows that just before C^i_{j-1} a successful `SC` on S is executed. Let SC^i_j be this `SC` instruction and let LL^i_j be its matching `LL` instruction. Let T^i_j be the read of $Toggles$ that is executed between LL^i_j and SC^i_j by the same thread.

Lemma 4 states that T^i_j is performed at the proper timing and returns the anticipated value.

▶ **Lemma 4.** *Consider any j, $0 < j \leq m_i$, it holds that T^i_j is executed after Q^i_j and reads $j \mod 2$ in $Toggles[i]$.*

Proof. Assume, by the way of contradiction, that T^i_j is executed before Q^i_j. Let π_x be the `Attempt` that executes T^i_j.

Assume first that $j = 1$. Then, by its definition, SC^i_1 (which is executed by π_x after T^i_1) writes to $S \rightarrow applied[i]$ a value equal to $\lceil j/2 \rceil \mod 2$; the code (lines 14, 18) implies that, in this case, T^i_1 reads 1 in $Toggles[i]$. Lemma 2 implies that $Toggles[i] = 0$ between C_0 and Q^i_1. Thus, T^i_1 could not read 1 in $Toggles[i]$, which is a contradiction.

Assume now that $j > 1$. By our assumption that T^i_j is executed before Q^i_j, it follows that LL^i_j, which is executed before T^i_j, precedes Q^i_j. In case that T^i_j follows Q^i_{j-1}, Lemma 2 implies that T^i_j reads $(j-1) \mod 2 \neq j \mod 2$ in $Toggles[i]$. By the pseudocode (lines 14, 18 and 56), it follows that π_x writes the value $(j-1) \mod 2$ into $S.applied[i]$. By its definition, SC^i_j stores $j \mod 2$ into $S.applied[i]$, which is a contradiction. Thus, T^i_j is executed before Q^i_{j-1}. By its definition, π^i_{2j-3} starts its execution after Q^i_{j-1} and finishes its execution before C^i_j. Lemma 1 implies that at least two successful `SC` instructions are executed in the execution interval of π^i_{2j-3}. Recall that LL^i_j precedes T^i_j and therefore also the beginning of π^i_{2j-3}, while by definition SC^i_j follows the end of π^i_{2j-3}. It follows that SC^i_j is not a successful `SC` instruction, which is a contraction. ◀

We next argue that, between certain configurations (namely C^i_{j-1} and C^i_j), the value of $S.applied[i]$ has the anticipated value and this value does not change in the execution interval defined by the two configurations.

▶ **Lemma 5.** *Consider any j, $0 < j \leq m_i$. At each configuration C between C^i_{j-1} and C^i_j, it holds that $S.applied[i] = (j-1) \mod 2$.*

Proof. Assume, by the way of contradiction, that there is at least one configuration between C^i_{j-1} and C^i_j such that $S \to applied[i]$ is equal to some value $v_x \neq (j-1) \mod 2$. Let C_x be the first of these configurations. Since only `SC` instructions of line 56 write on base object S, it follows that there is a successful `SC` instruction, let it be SC_x, executed just before C_x that stores v_x at $S.applied[i]$. Let π_x be the `Attempt` that executes SC_x and let T_x be the read instruction that π_x executes on line 14 of the pseudocode. By the definition of C^i_{j-1} and Q^i_{j-1}, it is implied that C^i_{j-1} follows Q^i_{j-1} and precedes Q^i_j. Lemma 2 implies that $Toggles[i] = (j-1) \mod 2 \neq v_x$ in any configuration between Q^i_{j-1} and Q^i_j. Since SC_x writes v_x into $S.applied[i]$, the pseudocode (lines 14 and 56) imply that T_x precedes Q^i_{j-1}. It follows that LL_x precedes Q_{j-1}, since LL_x precedes T_x. Therefore LL_x precedes C_{j-1}. This implies that there is a successful `SC` instruction, which is SC^i_{j-1}, between LL_x and SC_x. Thus, SC_x is a failed `SC` instruction, which is a contradiction. ◀

By Lemma 5 and the pseudocode (line 17), it follows that $S.papplied[i] = 1 - (j \mod 2)$ at C^i_j. Denote by $\tilde{C}^i_j$ the first configuration after C^i_j such that a successful `SC` instruction is executed.

The next lemma studies properties of $\tilde{C}^i_j$.

▶ **Lemma 6.** *$\tilde{C}^i_{j-1}$ precedes C^i_j and follows C^i_{j-1}.*

We next argue that the *applied* and *papplied* arrays of S indicate that p_i does not have a pending request between $\tilde{C}^i_{j-1}$ and C^i_j.

▶ **Lemma 7.** *$S.papplied[i] = S.applied[i]$ in any configuration between $\tilde{C}^i_{j-1}$ and C^i_j (C^i_j is not included).*

By Lemma 7, and by line 17, it follows that $S.papplied[i] = 1 - S.applied[i]$ at C^i_j. This and the definition of $\tilde{C}^i_j$ imply:

▶ **Lemma 8.** *$S.papplied[i] = 1 - S.applied[i]$ in any configuration between C^i_j and $\tilde{C}^i_j$ ($\tilde{C}^i_j$ is not included).*

We continue to define what it means for a process to apply a request on the simulated object. We say that a request *req* by some thread p_i *is applied* on the simulated object if (1) the `Read` instruction on $Toggles$ (line 14), executed by some request req' (that might be *req*

or any other request), includes p_i in the set of threads it returns, (2) procedure `Attempt`, executed by req' reads in $Announce[i]$, the request type written there by p_i for req and considers it as the new request type for p_i, (3) `Attempt` by req' calls `apply` for req (lines 22 - 45), and the execution of the `SC` at line 56 (let it be SC_r) on S succeeds. When these conditions are satisfied, we sometimes also say that req' applies req on the simulated object or that SC_r applies req on the simulated object.

▶ **Lemma 9.** *req^i_j is applied to the simulated object at configuration C^i_{3j-1}.*

Proof. Let p_h be the `Attempt` that executes the successful `SC` instruction (let it be SC_h this `SC` instruction) just before $\tilde{C}^i_j$. Let LL_h be the matching `LL` of SC_h. Since, SC_h is a successful `SC` instruction, it is implied that LL_h follows C^i_j. Observation 8 implies that LL_h reads for $S.applied[i]$ a value different from that stored in $S.papplied[i]$. Therefore, the if statement of line 21 returns *true*. Thus, a request for thread p_i is applied at $\tilde{C}^i_j$. Let req' be this request and assume, by the way of contradiction, that $req' \neq req^i_j$. Lemma 4 implies that π_h executes its read T_h on $Toggles$ after Q^i_j. By the pseudocode (lines 14, 22), π_h reads $Announce[i]$ after T_h, thus the reading of $Announce[i]$ by π_h is executed between Q^i_j and $\tilde{C}^i_j$. Since req^i_j writes its request to $Announce[i]$ before Q^i_j, the reading of $Announce[i]$ by π_h returns req^i_j. Thus, π_h applies req^i_j as the request of p_i in the simulated object. ◀

We are now ready to assign linearization points. For each $i \in \{1, ..., n\}$ and $j \geq 1$, we place the linearization point of req^i_j at $\tilde{C}^i_j$; ties are broken by the order imposed by identifiers of threads.

It is not difficult to argue that the linearization point of each request is placed in the execution interval of the request.

▶ **Lemma 10.** *Each request req^i_j is linearized within its execution interval.*

To prove consistency, denote by SC_l the l-th successful SC instruction on base object S. Let it_i be any iteration of the for loop of line 11 that is executed by a thread p_i. Let $SV_r(it_i)$ be the sequence of base objects read by the `LL` instructions of line 37 in it_i. Denote by $|SV_r(it_i)|$ the number of elements of $SV_r(it_i)$.

For each $1 \leq j \leq |SV_r(it_i)|$, denote by $SV^j_r(it_i)$ the prefix of $SV_r(it_i)$ containing the j first elements of $SV_r(it_i)$, i.e. $SV^j_r(it_i) = \langle sv^1_r(it_i), \ldots, sv^j_r(it_i)\rangle$, where $sv^j_r(it_i)$ is the j-th `LL` instruction performed by it_i on any base object. Let $SV^0_r(it_i) = \lambda$ be the empty sequence.

Let $V_r(it_i)$ be the sequence of insertions in directory D (lines 38-39) by it_i. Denote by $|V_r(it_i)|$ the number of elements of $V_r(it_i)$. Obviously, it holds that $|SV_r(it_i)| = |V_r(it_i)|$. For each $1 \leq j \leq |V_r(it_i)|$, denote by $v^i_r(it_i)$ the prefix of $V_r(it_i)$ containing the j first elements of $V_r(it_i)$, i.e. $V^j_r(it_i) = \langle v^1_r(it_i), \ldots, v^j_r(it_i)\rangle$, where $v_j(it_i)$ is the j-th value inserted to directory D. Let $V^0_r(it_i) = \lambda$ be the empty sequence.

Let $SV_w(it_i)$ be the sequence of shared base objects accessed by it_i while executing lines 51-52 (we sometimes abuse notation and say that a code line is executed by it_i to denote that the code line is executed by p_i during the execution of p_i). Denote by $|SV_w(it_i)|$ the number of elements of $SV_w(it_i)$. For each $1 \leq j \leq |SV_w(it_i)|$, denote by $SV^j_w(it_i)$ the prefix of $SV_w(it_i)$ that contains the j last elements of $SV_w(it_i)$, i.e. $SV^j_w(it_i) = \langle svw_1(it_i), \ldots, svw_j(it_i)\rangle$, where $svw_j(it_i)$ is the j-th request (lines 51-52) by it_i. Let $SV^0_w(it_i) = \lambda$ be the empty sequence.

Let $SV_a(it_i)$ be the sequence of shared base objects allocations during it_i iteration (lines 23-30). Denote by $|SV_a(it_i)|$ the number of elements of $SV_a(it_i)$. For each $1 \leq j \leq |SV_a(it_i)|$, denote by $SV^j_a(it_i)$ the prefix of $SV_a(it_i)$ that contains the j first elements of $SV_a(it_i)$, i.e. $SV^j_a(it_i) = \langle sva_1(it_i), \ldots, sva_j(it_i)\rangle$, where $sva_j(it_i)$ is the j-th base object allocation by it_i.

Let $SV_{arw}(it_i)$ be the sequence of allocations/reads/writes that it_i performs on base objects in lines 23-53 of the pseudocode. Denote by $|SV_{arw}(it_i)|$ the number of elements of $SV_{arw}(it_i)$. Obviously, it holds that $|SV_{arw}(it_i)| = |SV_a(it_i)| + |SV_r(it_i)| + |SV_w(it_i)|$. For each $1 \leq j \leq |SV_{arw}(it_i)|$, denote by $SV^j_{arw}(it_i)$ the prefix of $SV_{arw}(it_i)$ that contains the j first elements of sequence $SV_{arw}(it_i)$ (i.e. $SV^j_{arw}(it_i) = \langle svarw_1(it_i), \ldots, svarw_j(it_i)\rangle$) where $svarw_j(it_i)$ is the j-th base object allocations/reads/writes of base objects performed by it_i.

The next lemma states that for any process p_i that has a pending request, the i-th element of the *Announce* array stores the pending request of p_i for an appropriate time interval.

▶ **Lemma 11.** *Let $l > 0$ be any integer such that $S.applied[i] \neq S.papplied[i]$ at configuration C_{l-1}. Let req^i_j be the value of $Announce[i]$ at C_{l-1}. In any configuration between C_{l-1} and C_l, it holds that $Announce[i] = req^i_j$.*

▶ **Lemma 12.** *Let r be any shared base object other than S. For any $l > 0$, the following claims are true:*

1. *At most one successful* `SC` *instruction is executed on r between C_{l-1} and C_l.*
2. *In case that a successful* `SC` *instruction SC_w is executed on r, it holds that $r.seq < l$ just before SC_w and $r.seq = l$ just after SC_w.*
3. *Let it_i be some iteration of the loop of line 11 executed by a thread p_i that executes at least one successful* `SC` *instruction SC_r on r. If LL_r is the* `LL` *instruction of line 13 executed by it_i, then LL_r is executed after C_{l-1}.*
4. *Let it_i, $it_{i'}$ be two iterations of the for loop of line 11 executed by threads p_i and $p_{i'}$ respectively, such that that both it_i, $it_{i'}$ execute their* `LL` *instructions of line 13 somewhere between C_{l-1} and C_l, $l > 0$, and $|SV_{arw}(it_i)| \geq |SV_{arw}(it_{i'})|$. If both it_i, $it_{i'}$ execute line 49, just before C_l it holds that $SV_{arw}(it_i) = SV_{arw}(it_{i'})$.*

Proof. We prove the claims by induction on l. Fix any $l \geq 1$ and assume that the claims hold for l. We prove that the claims hold for $l + 1$.

We first prove Claim 1. Let SC' be the first of the successful `SC` instruction on r between C_{l-1} and C_l. We prove that $r.seq = l$ just after the execution of SC'. Assume by the way of contradiction that $r.seq = l' \neq j$. Let it_h be the iteration of line 13 executed by some thread p_h that executes SC'. Let LL' be the matching `LL` instruction of SC'. Since it_i executes successfully line 52 of the pseudocode, the pseudocode (lines 48 and 52) implies that the `VL` instruction of line 48 returns *true*. Since LL' is executed by it_i before this `VL` instruction, it follows that LL' precedes $SC_{j'}$. Thus, the `VL` instruction of line 48 is executed before $SC_{j'}$. Let $it_{i'}$ be the iteration of the loop of line 13 at which $SC_{j'}$ is executed and let $p_{i'}$ be the thread that executes $SC_{j'}$. Obviously, $LL_{j'}$ has been executed between $C_{l'-1}$ and $C_{l'}$. Since LL' is also executed between $C_{l'-1}$ and $C_{l'}$, the induction hypothesis (Claim 2.ii) implies that $SV_w(it_h) = SV_w(it_q)$. Thus, it_q has also executed an `SC` instruction on r. By lines 37, 49-52 and 56 of the pseudocode, it follows that there is a successful `SC` instruction on r between $SC_{l'-1}$ and $SC_{l'}$. Let SC_r be this instruction. By induction hypothesis (claim 1), it follows that $r.seq = j'$ just after the execution of SC_r. Since SC' is a successful `SC` instruction, LL' follows SC_r. By the pseudocode (lines 51-52), it follows that SC' is not executed, which is a contradiction. Therefore $r.seq = j$ just after the execution of SC_r. We now prove that there is no other successful `SC` instruction between SC' and C_l on r. Assume by the way of contradiction that at least one successful `SC` instruction takes place between SC' and C_l. Let SC'' be the first of these instructions. Since, SC'' is a successful `SC` instruction, it follows that its matching `LL` instruction LL'' follows SC'. By the pseudocode (lines 51-52), it follows that SC'' is not executed since $r.seq = S.seq$, which is a contradiction.

Claim 2 is proved using a similar argument as that above for Claim 1.

We now prove Claim 3. Assume by the way of contradiction that LL_p is executed between $SC_{j'-1}$ and $SC_{j'}$, $j' < j$. Let p_i be the thread that executes $SC_{j'}$ on some iteration it_i. By Claim 1 and by Claim 2, it follows that $r.seq \leq j'$ just before $SC_{j'}$. Thus SC_r is not executed, which is a contradiction. Thus, Claim 3 holds.

To prove Claim 4, it is enough to prove that $svarw_{l'}(it_i) = svarw_{l'}(it_{i'})$, for any $l' \leq |SV_{arw}(it_i)|$. We prove this claim by induction on the number $l' \leq |SV_{arw}|$ of elements of $SV_{arw}(it_i)$ (see appendix). ◀

Denote by α_i, the prefix of α which ends at SC_i and let C_i be the first configuration following SC_i. Let α_0 be the empty execution. Denote by l_i the linearization order of the requests in α_i.

We are now ready to prove that a_i is linearizable. This require to prove that the object state is consistent after the execution of each successful SC on S.

▶ **Lemma 13.** *For each $i \geq 0$, the following claims hold:*

1. *object's state is consistent at C_i, and*
2. *α_i is linearizable.*

Proof. We prove the claim by induction on i. The claim holds trivially; we remark that α_i is empty in this case. Fix any $i > 0$ and assume that the claim holds for $i - 1$. We prove that the claim holds for i.

By the induction hypothesis, it holds that: (1) object's state is consistent at C_{i-1}, and (2) α_{i-1} is consistent with linearization l_{i-1}. Let req be the request that executes SC_i. If req applies no request on the simulated object, the claim holds by induction hypothesis. Thus, assume that req applies $j > 0$ requests on the simulated object. Denote by $req_1, ..., req_j$ the sequence of these requests ordered with respect to the identifiers of the threads that initiate them.

Notice that req performs LL_i after C_{i-1} since otherwise SC_i would not be successful. By the induction hypothesis, object's is consistent at C_{i-1}. By Lemma 7, Observation 8, Lemma 9, and of the definition of $\tilde{C}^i_j$, it follows that each request req is applied exactly once. Thus, Lemma 12 imply that all threads that are trying to apply a set of requests between C_{i-1} and C_i do the following (1) apply the same set of requests with the same order, (2) all read the same consistent state of the object, (3) write the same set of base objects with the same values (although only one write succeeds), and (4) none of $req_1, \ldots, req_j$ have been applied in the past.

Given that $req_1, ..., req_j$ are executed by req sequentially, the one after the other in the order mentioned above, it is a straightforward induction to prove that (1) for each f, $0 \leq f \leq j$, request req_f returns a consistent response; moreover, $S \to st$ is consistent and once line 14 has been executed by req for all these requests. Therefore, $S \to st$ is consistent after the execution of req's successful SC. This concludes the proof of the claim. ◀

Lemma 13 implies that L-UC is linearizable. The discussion in Section 3.3 implies that L-UC is also wait-free and its step complexity is $O(n + kw)$. Thus:

▶ **Theorem 14.** *L-UC is a linearizable, wait-free implementation of a universal object. The number of shared memory accesses performed by L-UC is $O(n + kw)$.*

References

1. Yehuda Afek, Dalia Dauber, and Dan Touitou. Wait-free made fast. In *Proceedings of the 27th ACM Symposium on Theory of Computing*, pages 538–547, 1995.
2. James H. Anderson and Mark Moir. Universal constructions for multi-object operations. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 184–193, 1995.
3. James H. Anderson and Mark Moir. Universal Constructions for Large Objects. *IEEE Transactions on Parallel and Distributed Systems*, 10(12):1317–1332, December 1999.
4. Greg Barnes. A method for implementing lock-free shared data structures. In *Proceedings of the 5th ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270, 1993.
5. Phong Chuong, Faith Ellen, and Vijaya Ramachandran. A universal construction for wait-free transaction friendly data structures. In *Proceedings of the 22nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 335–344, 2010.
6. Andreia Correia, Pedro Ramalhete, and Pascal Felber. A Wait-Free Universal Construct for Large Objects, 2019. `arXiv:1911.01676`.
7. Panagiota Fatourou and Nikolaos D. Kallimanis. The RedBlue Adaptive Universal Constructions. In *Proceedings of the 23rd International Symposium on Distributed Computing*, pages 127–141, 2009.
8. Panagiota Fatourou and Nikolaos D. Kallimanis. A Highly-Efficient Wait-Free Universal Construction. In *Proceedings of the 23nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 325–334, 2011.
9. Panagiota Fatourou and Nikolaos D. Kallimanis. Revisiting the combining synchronization technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 257–266. ACM, 2012.
10. Panagiota Fatourou and Nikolaos D Kallimanis. Highly-Efficient Wait-Free Synchronization. *Theory of Computing Systems*, pages 1–46, 2013.
11. Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13:124–149, January 1991.
12. Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, November 1993.

Toward Linearizability Testing for Multi-Word Persistent Synchronization Primitives

Diego Cepeda
Department of Electrical and Computer Engineering, University of Waterloo, Canada
dcepeda@uwaterloo.ca

Sakib Chowdhury
Department of Electrical and Computer Engineering, University of Waterloo, Canada
sm6chowd@uwaterloo.ca

Nan Li
Department of Electrical and Computer Engineering, University of Waterloo, Canada
n69li@uwaterloo.ca

Raphael Lopez
Department of Mechanical and Mechatronics Engineering, University of Waterloo, Canada
rwlopez@uwaterloo.ca

Xinzhe Wang
Department of Electrical and Computer Engineering, University of Waterloo, Canada
x793wang@uwaterloo.ca

Wojciech Golab
Department of Electrical and Computer Engineering, University of Waterloo, Canada
wgolab@uwaterloo.ca

Abstract

Persistent memory makes it possible to recover in-memory data structures following a failure instead of rebuilding them from state saved in slow secondary storage. Implementing such recoverable data structures correctly is challenging as their underlying algorithms must deal with both parallelism and failures, which makes them especially susceptible to programming errors. Traditional proofs of correctness should therefore be combined with other methods, such as model checking or software testing, to minimize the likelihood of uncaught defects. This research focuses specifically on the algorithmic principles of software testing, particularly linearizability analysis, for multi-word persistent synchronization primitives such as conditional swap operations. We describe an efficient decision procedure for linearizability in this context, and discuss its practical applications in detecting previously-unknown bugs in implementations of multi-word persistent primitives.

2012 ACM Subject Classification Software and its engineering → Synchronization; Theory of computation → Shared memory algorithms; Computer systems organization → Reliability

Keywords and phrases Shared memory, persistent memory, synchronization, multi-word primitives, concurrency, correctness, software testing

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.19

Funding *Diego Cepeda*: Consejo Nacional de Ciencia y Tecnología
Raphael Lopez: University of Waterloo President's Research Award
Xinzhe Wang: University of Waterloo President's Research Award
Wojciech Golab: Natural Sciences and Engineering Research Council (NSERC) of Canada, Discovery Grants Program; Ontario Early Researcher Award; Google Faculty Research Award

© Diego Cepeda, Sakib Chowdhury, Nan Li, Raphael Lopez, Xinzhe Wang, and Wojciech Golab;
licensed under Creative Commons License CC-BY
23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller; Article No. 19; pp. 19:1–19:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Persistent memory makes it possible to recover in-memory data structures following a failure instead of rebuilding them from state saved in slow secondary storage. Implementing such recoverable data structures correctly is challenging as their underlying algorithms must provide both effective concurrency control to harness multi-core parallelism, and recovery procedures to ensure that failures (e.g., process or system crashes) do not corrupt data. The complex states and state transitions of these algorithms make recoverable structures much harder to analyze using traditional means such as rigorous proofs of correctness, particularly with the consideration of failures and subsequent recovery adding a new dimension of difficulty. Automated model checking tools such as PlusCal/TLA+ [23, 24] can be helpful in this context but are difficult to use due to lack of native support for modeling access to persistent memory.

This research aims to augment traditional analysis of correctness with a gray-box software testing approach in which execution histories of synchronization primitives are generated empirically and checked for correctness using a rigorous decision procedure. The correctness criterion under consideration is Herlihy and Wing's widely-adopted *linearizability* property [19], which states that operations applied to an object by a collection of threads must behave as though they take effect instantaneously at some point between their invocation and response events. Deciding linearizability given a history of operations is NP-hard for many data structure types, and often becomes tractable under certain simplifying assumptions [15]. Notably, this holds for histories of primitive read, write, and swap (i.e., Fetch-And-Store) operations if the *reads-from mapping* is known, meaning that the value returned by each operation is either the initial value or the value assigned by a unique write or swap operation, which makes the problem solvable in quasilinear time.

This paper extends and enhances prior work on deciding linearizability for synchronization primitives, such as swaps and atomic counters, as follows:

1. We extend the algorithmic foundations of linearizability testing to multi-word read and conditional swap (i.e., Compare-And-Swap) operations, which are important building blocks of practical data structures for persistent memory.
2. Our techniques take into account operations that are interrupted by failures and do not produce a response, which makes it difficult to deduce their effect.
3. We present an empirical study of a software tool that implements our analysis techniques. The tool is applied to two codebases, and successfully detects previously unknown bugs in both, including one bug that is related directly to the use of persistent memory. We also evaluate the scalability of the tool, and demonstrate that its running time grows nearly linearly with the size of the input history.

2 Model

The model is based closely on Herlihy and Wing's [19]. There are n asynchronous processes, labeled $p_1, \ldots, p_n$, that interact by applying operations on a shared memory with a well-defined initial state. Two types of operations are permitted: an atomic multi-word Read (MwR), and an atomic multi-word Compare-And-Swap (MwCAS) operation. MwCAS is the atomic execution of the pseudocode shown in Figure 1. We assume somewhat unconventionally that the return value of an unsuccessful MwCAS indicates the responsible memory location, which is a crucial piece of information exploited by our gray-box technique (see Section 5.2). Software simulations of multi-word CAS (e.g., [12, 18, 30]) that return a Boolean can be modified easily to meet this specification.

Procedure MwCAS($m_1, \ldots, m_k$: memory locations; $e_1, \ldots, e_k$: expected values; $n_1, \ldots, n_k$; new values)

```
1 old_i := *m_i for all i, 1 ≤ i ≤ k
2 if old_i = e_i for all i, 1 ≤ i ≤ k then
3     *m_i := n_i for all i, 1 ≤ i ≤ k
      // successful CAS
4     return 0
5 else
      // unsuccessful CAS
6     return m_i such that old_i ≠ e_i
```

Figure 1 Sequential specification of multi-word Compare-And-Swap (MwCAS).

Actions of processes are represented using a *history*, which is a sequence of *steps*. Each step is either the invocation of a shared memory operation, or the response of a previously invoked operation. An invocation step is of the form INV(p_i, op, $args$) where p_i is a process, op is the name of an operation, and $args$ are its arguments including the set of memory locations accessed. A response step is of the form RES(p_i, ret) where p_i is a process, and ret is the return value of the operation. A response step s_r is *matching* with respect to an invocation step s_i in a history H if the following criteria are met: (i) s_r and s_i refer to the same process; (ii) s_i precedes s_r in H; and (iii) no other step by p_i occurs between s_i and s_r in H. Given a history H, its projection onto the steps of a process p_i is denoted by $H|p_i$. A history is *sequential* if it is either empty, or a non-empty sequence of alternating invocation and response steps where each invocation is followed immediately by a matching response. A history H is *well-formed* if, for every process p_i, the projection $H|p_j$ is sequential.

An *operation* in a history H is a pair of steps comprising an invocation and a matching response. Given distinct operations op_1 and op_2, we say that op_1 *happens before* op_2 in H (denoted $op_1 <_H op_2$) if the response step of op_1 precedes the invocation step of op_2 in H. Operations op_1 and op_2 are *concurrent* if neither $op_1 <_H op_2$ nor $op_2 <_H op_1$. A well-formed history H is *linearizable* if there exists a sequential history S, called the *linearization of* H, satisfying the following properties: (i) for each process p_i, $H|p_i = S|p_i$; (ii) for any distinct operations op_1 and op_2 in H, if $op_1 <_H op_2$ then $op_1 <_S op_2$; and (iii) S is *legal*, meaning that operations in S produce responses according to their sequential specification (e.g., Figure 1).

For simplicity of presentation and due to lack of space, we omit from the model the more general definition of linearizability for pending or incomplete operations (i.e., ones lacking responses), and its extensions for persistent memory [2, 7, 17, 21]. In Section 5.3, we will adopt the approach of transforming a given history H to a well-formed history H' such that H has the consistency property under consideration if and only if H' does.

3 Background

The problem of deciding linearizability given a history of operations has been studied widely in the context of atomic read/write registers, and is known to be NP-complete in the general case [15]. It is solvable in polynomial time for reads, writes, and swaps under the assumption that a *reads-from mapping* is known, meaning that each read returns either the initial value at a memory location or the value assigned to this location by a unique write. Such a relation can be established easily for the purpose of software testing by generating execution histories

using a driver program that embeds distinct tags (e.g., based on process IDs and per-process counters) in the values written to shared memory.

Assuming that the reads-from relation exists for a history H, our goal is to first decide whether H is linearizable, and if it is not, identify specific anomalies – small subsets of the operations in H that conspire to violate linearizability. Consider the example history illustrated in Figure 2 (a), where a shared register object x is accessed by three writes that assign values 1,2,3, followed by two reads that return 1 and 3. The barbell symbols represent the time intervals of operations, and process IDs are omitted. The history shown is not linearizable because x is overwritten twice between the write and read of value 1, which are denoted by $W(x, 1)$ and $R(x) \rightarrow 1$.

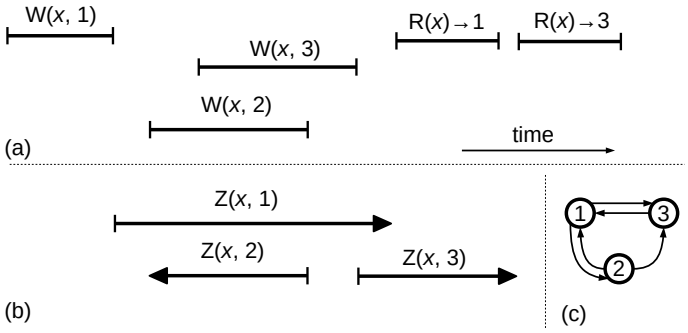


Figure 2 Example execution history (a), its zone-based representation (b), and its graph-theoretic representation (c).

Procedures for deciding linearizability come in two flavours: graph-based, and zone-based. In both cases, the history is first checked for obvious anomalies such as *dangling reads*, which return a value that was never written and is different from the initial value, and *read-write inversions*, where the read of some value v happens before the write of v. Once such cases are ruled out, more subtle anomalies are analyzed. In the graph-based approach inspired by Misra's Axioms for memory access [26], a *precedence graph* $G(V, E)$ is defined whose vertices represent the values read or written, and where an edge (v_1, v_2) exists whenever some operation op_1 that accesses v_1 happens before an operation op_2 that accesses a different value v_2. The input history is linearizable if and only if G has no directed cycles. The graph for our example history is shown in Figure 2 (c), and exhibits several such cycles.

In the zone-based approach of Gibbons and Korach [15], each value v is represented using a time interval called a *zone*, which spans from the time of the earliest response step to the time of the latest invocation step of any operation that reads or writes v. If the earliest response precedes the latest invocation, a *forward zone* occurs, otherwise a *backward zone occurs*. Intuitively, a forward zone for v is a minimal interval of time for which v is continuously the current value, and a backward zone for v is an interval of time containing at least one point at which v is the current value. The history is linearizable if no two forward zones overlap, and no backward zone is contained entirely within a forward zone. The zones for our running example are shown in Figure 2 (b). The overlap between forward zones $Z(x, 1)$ and $Z(x, 3)$, and similarly the containment of backward zone $Z(x, 2)$ within $Z(x, 1)$, indicate that the history is not linearizable.

4 Graph-Theoretic Formulation

Our approach to analyzing the linearizability of persistent synchronization primitives is based on precedence graphs, similarly to the technique described informally in Section 3 based on Misra's Axioms [26]. The key advantage of the graph-based approach is that it represents the constraints on the linearization order in an intuitive way, and (as we show in this paper) can be generalized beyond single-word read/write registers to accommodate a variety of multi-word synchronization primitives. On the other hand, its main weakness is the potentially large number of edges required to construct the graph. For example, given a sequential history with n operations, the precedence graph has $\Theta(n)$ vertices and $\Theta(n^2)$ edges in the worst case since every pair of operations is related by the happens before relation; this can make linearizability analysis quite slow for large execution histories. In comparison, the zone-based algorithm of Gibbons and Korach [15] runs in $O(n \log n)$ time, but is more difficult to analyze and does not generalize easily to multi-word primitives.

Our graph-theoretic approach builds on the simple algorithm described in Section 3, and is inspired by [3, 26]. Given a history H, we construct a *precedence graph* $G(V, E)$ whose vertices represent individual operations rather than the values accessed, and the directed edges represent constraints on the order in which certain pairs of operations must appear in all possible linearizations. A special *genesis vertex* is added to V to represent the initial state of shared memory, and can be regarded as a multi-word write (MwW) operation that creates this state. The precedence graph G is a multigraph, meaning that each pair of vertices can be connected by more than one directed edge. The edge multiset E includes an edge (op_1, op_2) in the following scenarios:

1. *Happens-before edge*: indicates that $op_1 <_H op_2$.
2. *Reads-from edge*: indicates a read-after-write dependency, meaning that op_1 is a multi-word write (genesis vertex) or a successful MwCAS that writes a value v to some memory location m, and op_2 is a multi-word read or successful MwCAS that reads v from m.
3. *Auxiliary edge*: indicates any other precedence constraint on op_1 and op_2.

As explained later on in Section 5, we use two types of auxiliary edges. If op_2 is a successful MwCAS that changes the value at some memory location m from v to v', then the edge (op_1, op_2) indicates a write-after-read dependency where op_1 is a multi-word read that reads v from m. If op_2 is an unsuccessful MwCAS that expects to read a value v at some memory location m and instead encounters a value that is different from v (see Figure 1), then the edge (op_1, op_2) indicates a special type of read-after-write dependency where op_1 is a successful MwCAS that overwrites v at m with a different value.

Our graph-theoretic analysis technique is based on the following assumptions:

▶ **Assumption 1.** *For every history H, for every memory location m, and for every successful MwCAS operation op that writes some value v to m, v is different from the initial value at m and from any value written to m by another successful MwCAS operation.*

▶ **Assumption 2.** *For every history H, for every memory location m, and for every MwCAS operation op that accesses m, the value expected by op at m is a value that was read from m by some MwR or successful MwCAS operation op' such that $op' <_H op$.*

Assumption 1 helps to establish a reads-from mapping, and Assumption 2 simplifies reasoning about unsuccessful MwCAS operations. Both assumptions can be enforced by the tool (e.g., a benchmark program) used to generate histories for testing, without changing the implementation of the synchronization primitive under consideration.

Under the above assumptions, the problem of deciding linearizability given the precedence graph $G(V, E)$ is reduced to the problem of verifying the following structural properties of G:

(i) G is acyclic;

(ii) Every multi-word read or successful MwCAS operation has exactly one incoming reads-from edge for each memory location accessed, and every unsuccessful MwCAS operation has exactly one incoming auxiliary edge; and

(iii) For every memory location m, there exists a directed path of reads-from edges for m that starts at the genesis vertex, and visits every vertex representing a successful MwCAS operation that accesses m. (This path is unique when property i holds.)

Property i is necessary to ensure that all constraints on the linearization order are met, but is not sufficient by itself because operations that return incorrect responses can generate other types of structural anomalies (see Section 6.2). Properties ii–iii compensate for this by ensuring that the state observed by an operation can be attributed to a unique sequence of state transitions starting from the initial state. All three properties can be checked in time linear in the size of G (i.e., $O(|V| + |E|)$), assuming an adjacency list representation where each reads-from edge is labeled with the corresponding memory location m. Property i is established by running depth-first-search (DFS) on the entire graph, and ensuring that no back edges are present. Property ii is established by counting in-edges for the relevant vertices. Property iii is decided using a greedy algorithm that, for each memory location m, repeatedly follows reads-from edges for m starting from the genesis vertex. Supposing that properties i–ii have already been checked, the greedy algorithm either discovers the required path, or else reaches a fork, which proves that such a path does not exist because each vertex under consideration has exactly one incoming reads-from edge for m.

Procedure `ReduceHB` (H: input history)

```
Edges := ∅
S := set of operations in H
for any operation op in H, define Start(op) and Fin(op) as the position in H of op's
    invocation and response event, respectively
L := operations in S sorted in ascending order by Start(op)
for op in L do
    if op is the last operation in L then
        return Edges
    op' := earliest operation in L such that op <_H op'
    f := ∞
    L' := suffix of L from op' (inclusive) onward
    for op'' in L' do
        if Start(op'') > f then
            break from inner for loop
        else
            Edges := Edges ∪ {(op, op'')}
            f := min(f, Fin(op''))
return Edges
```

Figure 3 Computation of a minimal subset of happens-before edges.

Before presenting the specifics of read-from and auxiliary edges in Section 5 of the paper, we first describe a general optimization that can reduce the number of edges required to decide linearizability. This optimization is particularly relevant in our work for two reasons. First, our precedence graph uses vertices to represent operations rather than the values accessed by these operations, and tends to generate more edges than the technique presented earlier in Figure 2 (c) because several operations may access one value. Second, because our research emphasizes multi-word primitives, linearizability cannot be checked independently for each memory word, and this puts additional pressure on the algorithm to deal efficiently with large inputs.

The edge set of the precedence graph G tends to be dominated by happens-before edges, whose number in the worst case (i.e., when the input history is nearly sequential) grows quadratically with the number of operations. In contrast, the number of reads-from and auxiliary edges tends to grow linearly under Assumption 1. Our optimization aims to reduce the number of happens-before edges by exploiting the transitivity of the happens-before relation, which makes it possible to derive some precedence constraints indirectly from others. For example, if the input history H contains three operations op_1, op_2, op_3 such that $op_1 <_H op_2$ and $op_2 <_H op_3$, then there is no need to explicitly represent $op_1 <_H op_3$. Applying this observation, it is possible to compute a minimal subset of edges whose transitive closure is the entire happens before relation. The algorithm for selecting such a subset of edges is shown in Figure 3 as procedure `ReduceHB`. Its correctness properties are captured in Theorems 1–2, whose proofs are omitted due to lack of space.

▶ **Theorem 1.** *For any history H, let $G(V, E)$ be the graph whose vertices represent the operations in H, and where E is the set of edges output by* `ReduceHB(H)`*. Then for every pair of operations op_1, op_2 in H, $op_1 <_H op_2$ if and only if there is a directed path in G from op_1 to op_2. Furthermore, if H contains three operations op_1, op_2, op_3 such that $op_1 <_H op_2$ and $op_2 <_H op_3$, then E does not contain the edge (op_1, op_3).*

▶ **Theorem 2.** *Let C be the* point contention *for a given history H, which is the maximum number of operations that overlap at a single point in time. Then procedure* `ReduceHB(H)` *returns a set of $O(Cn)$ edges where n is the number of operations in H, and it has an implementation with time complexity $O(n \log n + Cn)$.*

Intuitively, procedure `ReduceHB` iterates over all operations in the input history H in increasing order of their start time (outer loop), and for each such operation op it identifies a maximal subset E_{op} of operations that succeed op directly in the partial order $<_H$ (inner loop). It suffices to create happens-before edges from op to each element of E_{op}, as any other operation op' that succeeds op in $<_H$ also succeeds some operation in E_{op}. The size of E_{op} is bounded by the point contention parameter C referred to by Theorem 2 because all pairs of operations in E_{op} are concurrent. For practical purposes, C is a small constant, for example the number of parallel threads used in an experiment to generate an execution history. Supposing that $C \in O(1)$, Theorem 2 implies $O(n \log n)$ running time (same as sorting) and a graph with $O(n)$ happens-before edges. As we will explain later on in Section 5, the precedence graph contains $O(kn)$ reads-from and auxiliary edges for k-word operations, and so our selection of happens-before edges ensures that the entire edge set has size $O(kn)$.

5 Constructing the Precedence Graph

In this section, we describe in more detail the formation of the precedence graph, focusing on the reads-from and auxiliary edges. We begin with a discussion of different variations of the multi-word swap, and assume initially that every operation has both an invocation and

matching response. In Subsection 5.3, we finally discuss how to handle operations that are interrupted by failures. We do not discuss (multi-word) write operations, but note that they can be incorporated into our framework fairly easily, and leave the details to future work.

5.1 Multi-Word Reads and Successful MwCAS Operations

For didactic purposes, we first describe the details of dealing with k-word reads and successful swaps only, where $k \geq 1$. Consider an operation $\text{MwR}(m_1 \dots m_k) \rightarrow r_1 \dots r_k$ that returns r_i from memory location m_i. The reads-from and auxiliary edges required are derived by generalizing Misra's Axioms [26] to multi-word operations. Specifically, the operation must satisfy the following criteria for each memory location m_i in any linearization L of the given history H, where op_R denotes the above multi-word read:

1. There exists a single operation op (possibly the one represented by the genesis vertex) that writes r_i to m_i, and such that $op <_L op_R$.
2. For any operation op' that writes a value $w_i \neq r_i$ to m_i, either $op' <_L op$ or $op_R <_L op'$.

Criteria 1 and 2 are both needed to ensure that L is legal, particularly that op_R returns a correct value for location m_i. In other words, op_R must return a value that was written to m_i, and moreover this must be the value assigned by the most recent update to m_i that precedes op_R in L. For criterion 1, structural property ii of the graph ensures that op exists, Assumption 1 ensures that op is unique, and the reads-from edge (op, op_R) encodes the constraint $op <_L op_R$. For criterion 2, we add auxiliary edges according to the following procedure: we identify for each memory location m the source vertex op of the incoming reads-from edge for m, and its immediate successor s on the path of reads-from edges referred to by structural property iii. If s exists, we insert an auxiliary edge (op_R, s), which ensures that op' in criterion 2 cannot be linearized between op and op_R.

For a successful MwCAS operation op_C, we have analogous criteria where op_R is replaced with op_C, and criterion 2 is relaxed to handle the case when $op' = op_C$. Reads-from edges are inserted as for MwR operations, but auxiliary edges are not used as otherwise one could form a loop from op_C back to itself. In this case, structural property iii compensates for the lack of auxiliary edges by ensuring that op_C has sufficient outbound reads-from edges.

An example of the dependency graph for a linearizable history is shown in Figure 4. For the sake of clarity, some edges as well as some operations required by Assumption 2 are omitted. This example satisfies structural properties i–iii. A non-linearizable example is then shown in Figure 5, where the swap operation for the state change $B : 1 \rightarrow 2$ happens before the read, which makes the response of the read operation stale with respect to memory location B (but not A). In this second example, the swap and the read lie on a cycle, which violates structural property i.

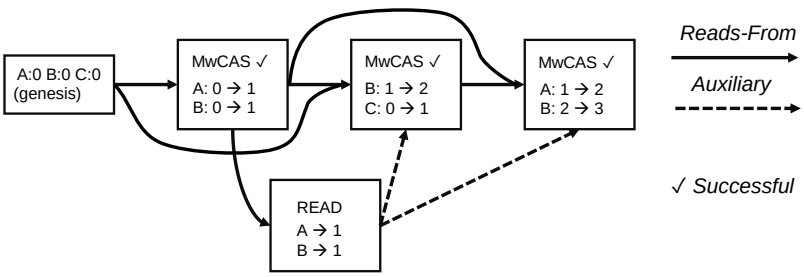


Figure 4 Simplified precedence graph for a linearizable history.

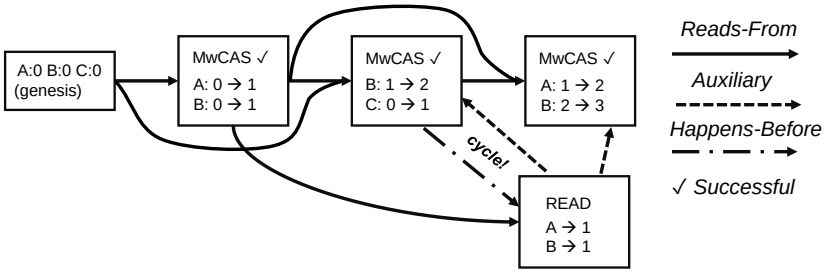


Figure 5 Simplified precedence graph for a non-linearizable history.

5.2 Incorporating Unsuccessful MwCAS Operations

Conditional swap operations present unique challenges that do not exist with unconditional swaps because they can take effect without modifying the state of shared memory. Depending on the implementation, such unsuccessful operations either return the (unexpected) values read, or simply signal that the swap failed. In the former case, the unsuccessful MwCAS is treated like a multi-word read, and so the technique from Subsection 5.1 is sufficient. In the latter case, we assume (as in Figure 1) that the response indicates a memory location m_i for which the observed value was different from the expected value e_i. Also, we require (see Assumption 2) that e_i was read from m_i before the MwCAS. Thus, given some operation op that writes e_i at m_i, the unsuccessful MwCAS$(m_1 \dots m_k, e_1 \dots e_k, n_1 \dots n_k)$ denoted by op_U requires that there exists some operation op' that writes $w_i \neq e_i$ at m_i and satisfies $op <_L op' <_L op_U$ in any linearization L of the given history H. For the purpose of building the dependency graph, we identify op' as the unique successor s to the vertex representing op on the path of reads-from edges for memory location m_i referred to by structural property iii, and insert an auxiliary edge (s, op_U). Intuitively, this encodes the unsuccessful MwCAS reading the value w_i assigned by op', or a newer value.

Figures 6 and 7 illustrate the technique of interpreting an unsuccessful MwCAS in this manner. In Figure 6, the unsuccessful MwCAS takes effect after B is swapped from 1 to 2, and before A is swapped from 1 to 2. Thus, it fails because of B. This is captured by the auxiliary edge from the MwCAS that swapped 2 into B, to the unsuccessful MwCAS. If the unsuccessful MwCAS instead takes effect after A is swapped from 1 to 2 then, as shown in Figure 7, the auxiliary edge closes a cycle, violating structural property i.

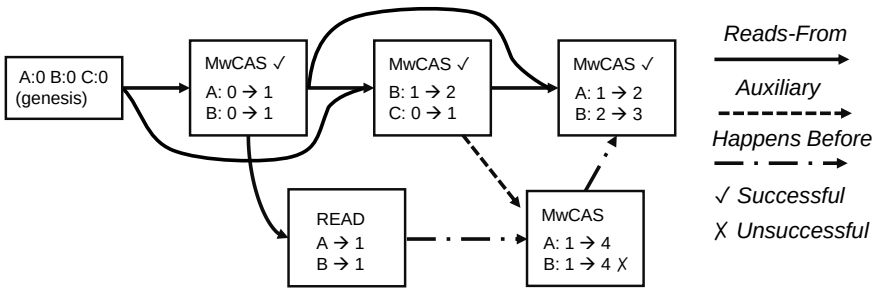


Figure 6 Simplified precedence graph for a linearizable history with an unsuccessful MwCAS that fails because of memory location B.

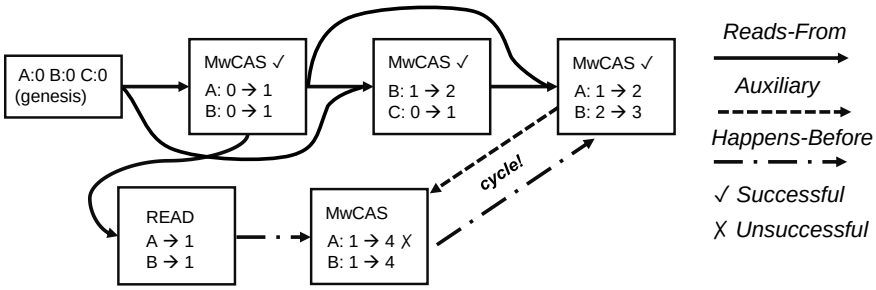


Figure 7 Simplified precedence graph for a non-linearizable history with an unsuccessful MwCAS that fails because of memory location A.

5.3 Operations Interrupted by Crashes

Herlihy and Wing [19] deal with incomplete operations by adding (judiciously chosen) matching responses to a subset of such operations and ignoring the rest. This technicality complicates the analysis of linearizability substantially, and so we have elected to assume in Section 2 that all operations are complete. We now describe how to transform the input history to achieve this property without affecting its linearizability. Regardless of which flavor of linearizability one considers [2, 7, 17, 19, 21], the following rules apply to histories of multi-word reads and MwCAS operations:

1. Incomplete reads can be excluded from the history as they cannot violate linearizability because their responses are not known.
2. An incomplete MwCAS can also be excluded provided that none of the new values it was attempting to swap in has been read by another operation. This holds whether or not the MwCAS actually succeeded and took effect.
3. An incomplete MwCAS whose effect was observed by another operation must be given a matching response. Moreover, the response must indicate that the MwCAS was successful.

The exact placement of the response step stipulated in clause 3 depends on the specific correctness property at hand. For strict linearizability [2], the matching response is placed immediately before the crash. For recoverable linearizability [7], the matching response is inserted at the end of the history, and an auxiliary edge is added from the completed operation to the next operation of the same process, if one exists.[1] For durable linearizability [21], only the matching response is added at the end of the history.

6 Evaluation

This section presents our evaluation of the linearizability analyzer's performance (Subsection 6.1) and effectiveness (Subsection 6.2). All experiments are conducted using a commodity server equipped with four Xeon E5-4620 2.20GHz CPUs. The system has 32 cores total and 256GB of DRAM, 17GB of which are reserved for emulating persistent memory. (The system lacks persistent memory.) The software environment includes Ubuntu Linux 18.04 LTS with kernel version 4.15.0-58-generic, gcc 7.4.0, and OpenJDK 11. Java is used to implement the linearizability analyzer, which comprises 1680 lines of single-threaded code, and can process

[1] Technically speaking, the auxiliary edge should be directed to the next operation applied by the same process on the *same object*. However, given that we are dealing with multi-word operations over a flat address space, we treat the entire collection of memory locations as one shared object.

execution histories obtained by instrumenting persistent synchronization primitives written in any programming language. The C++ compiler is used to compile two implementations of persistent atomic multi-word swap for linearizability analysis.

6.1 Performance Experiments

This section discusses the performance of the analyzer through empirical analysis along two dimensions. First, we assess scalability by measuring the running time of the analyzer on inputs of varying size. Then, we quantify the speed-up due to our optimized method of selecting happens-before edges (Figure 3).

Given the high performance of the synchronization primitives being analyzed, our gray box testing approach has a tendency to generate large execution histories. For example, experiments lasting only one minute can produce histories with millions of operations (n). As a result, it is imperative that running time of the analyzer grows nearly linearly with n, as opposed to a higher-degree polynomial. Naive generation of edges and their selection would instead result in running time and graph size growing quadratically (or worse) with n. As we show through experiment, judicious use of hash tables instead of nested loops, combined with the optimized happens-before edge selection algorithm from Section 4, avoids this problem.

Figure 8 shows the performance of the analyzer with the above optimizations. The analyzer was run repeatedly on increasingly long prefixes of a 1GB execution history log generated using a third-party multi-word Compare-And-Swap implementation (see *CodebaseA* in Section 6.2). Each run was performed 5 times. The average running times and standard deviations are shown in Figure 8. The entire log was deemed linearizable by the analyzer in this case, and same for its prefixes, as expected. Log size is measured in bytes rather than number of operations because both are highly correlated for uniform workloads (approx. 100 bytes per event, two events per operation). Figure 8 shows a definitively linear dependency of both the running time and number of edges on the log size, and hence on n. Minor variations are visible in the running times despite our attempts to control the environment, for example by disabling turbo-boost, which is expected for Java code due to factors such as garbage collection. The trendline still falls within a standard deviation of the runtime for all prefixes.

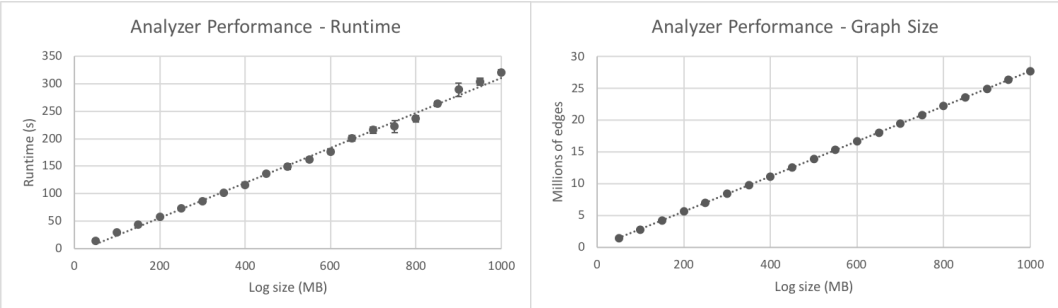


Figure 8 Running time of the analyzer in seconds over prefixes of a 1GB log file (left), and number of edges in the dependency graph constructed by the analyzer (right).

In the next experiment, we compare our optimized implementation of the analyzer against a naive baseline implementation that inserts all possible happens-before edges. Running time is measured over shorter prefixes of the input log used in the scalability experiment presented earlier in Figure 8. The new results are presented in Figure 9, which shows that the `ReduceHB` optimization from Section 4 indeed reduces the total number of edges in the dependency graph from quadratic to linear in the number of operations n. Similarly, the

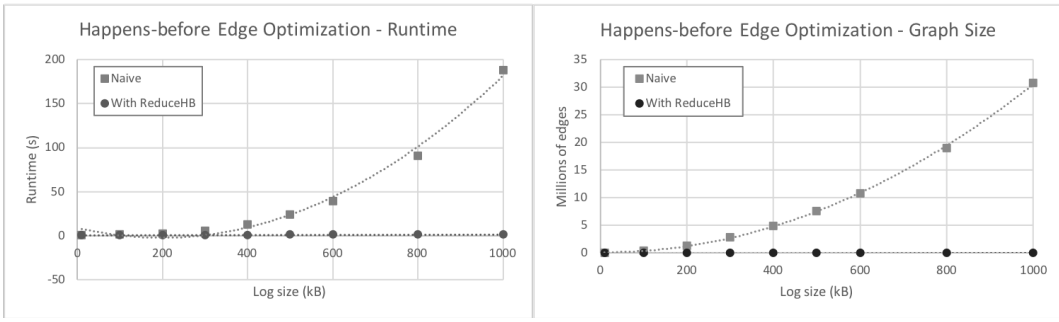


Figure 9 Running time comparison of analyzer with and without the `ReduceHB` optimization from Section 4 (left), and the size of the resulting dependency graphs (right).

running time is reduced to nearly linear. The 1 MB log file, for example, takes 100x longer to process without the optimization. Due to the nonlinearly increasing running time, testing the naive implementation on larger logs quickly became unfeasible.

In a third experiment, we measured the performance penalty due to the logging of execution histories. The results indicate a slowdown of roughly 10%.

6.2 Effectiveness

In this section, we comment on the effectiveness of our linearizability analyzer in the field. To that end, we present a case study in which the analyzer is applied to two implementations of atomic-multi-word swap primitives, in both cases revealing previously unknown bugs.

To test our analyzer, we selected two codebases for analysis: *CodebaseA* is an industrial-grade persistent multi-word compare-and-swap with an open-source implementation [30], and *CodebaseB* is a persistent multi-word unconditional swap developed by a research assistant. We first used a benchmark to generate failure-free single-threaded executions, and discovered no linearizability violations. Next, we considered failure-free concurrent executions with two threads, and detected linearizability violations in *CodebaseB*. As shown in Figure 10, thread t_0 applies an operation that successfully swaps the value i_2 into address A. After that, t_0 performs another swap on the same address A, changing its value successfully from i_2 to i_3. Thread t_1 attempts to swap address A concurrently, but the changes made by t_0 are not reflected in the value read by t_1, which is the previous value i_2 set by t_0 in its first operation. Figure 11 shows a section of the corresponding precedence graph, which violates structural property iii (see Section 4) as no path of reads-from edges for memory location A passes through all the vertices due to a fork.

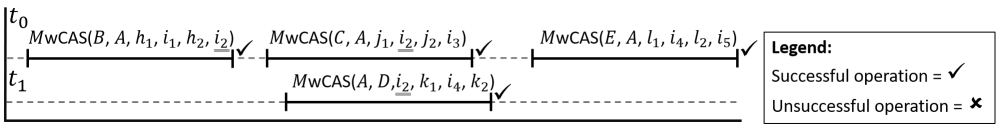


Figure 10 CodebaseB, non-linearizable execution history indicating a bug.

CodebaseA, showed no linearizability issues in failure-free runs with two threads, but generated non-linearizable histories with three threads. We found spurious events where a previously invoked unsuccessful MwCAS operation interferes with another MwCAS operation that should have succeeded (i.e., causes the latter to become unsuccessful). To further

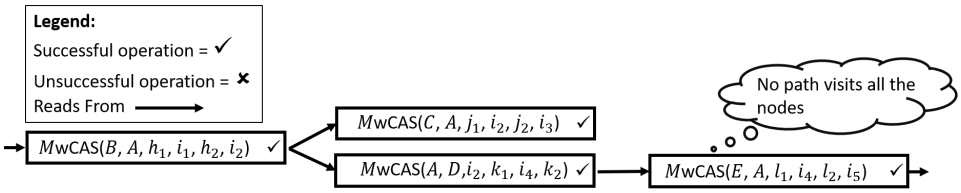


Figure 11 CodebaseB, section of precedence graph for the history shown in Figure 10.

understand the spurious events, refer to Figure 12. In this scenario, t_0 begins an operation on addresses B and A, and before t_0 finishes, t_1 begins an operation on addresses C and A. As t_0 finishes the multiword operation, we expect t_1's operation to be unsuccessful due to address A being changed by t_0. Prior to t_0 and t_1 completing their operations, t_2 begins an operation on C and D. Since the completed operation of t_0 does not interfere with the addresses accessed by t_2, and since the operation by t_1 which shares an address in common with t_2 is unsuccessful, we would expect t_2's operation to succeed, but this is not the case. Figure 13 shows a section of the corresponding precedence graph, which violates structural property ii (see Section 4) because the vertex for t_2's unsuccessful MwCAS is disconnected from the rest of the graph.

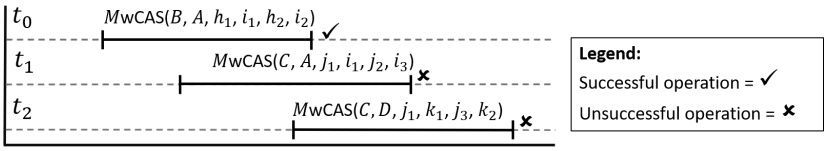


Figure 12 CodebaseA, non-linearizable execution history indicating a bug.

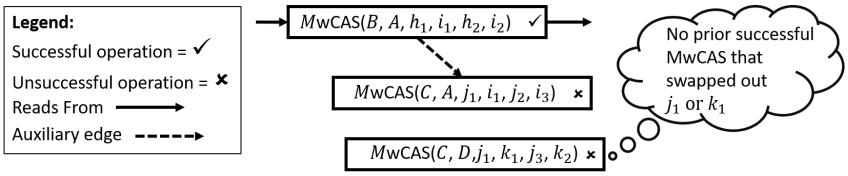


Figure 13 CodebaseA, section of precedence graph for the history shown in Figure 12.

Before explaining the root cause of the anomalous behavior, we explain briefly the design of *CodebaseA* [30]. MwCAS operations use two types of structures: operation descriptors and word descriptors. Operation descriptors record the arguments, the response of the operation to be performed, the status of the operation (e.g., undecided, successful, unsuccessful), as well as an array of word descriptors. The word descriptors contain the target word address, the expected value to compare against, the new value, and a back pointer to the MwCAS descriptor. The execution of an MwCAS has two phases. *Phase*1 installs a pointer to the MwCAS descriptor in all the target addresses, provided that the current value of a word matches the expected value. The execution path of *Phase*2 depends on the outcome of *Phase*1. For a successful *Phase*1, *Phase*2 installs the new values to the target addresses. For an unsuccessful *Phase*1, *Phase*2 resets any target word that points to the MwCAS descriptor back to its old value. In addition, the algorithm embeds a "dirty bit" in each word to mark data that has been updated but not yet flushed to persistent memory.

The bug in *CodebaseA* occurs when a thread attempts to install a word descriptor (see line 5 of Algorithm 2 in [30]) using a single-word CAS instruction, and encounters the expected value with the dirty bit set at one of the target memory locations. In Figure 12, such a value is observed by t_2 at location C, which is modified earlier and then restored by t_1. (The bug occurs when t_2 reads C before t_1's unsuccessful MwCAS has a chance to persist the restored value and clear the dirty bit.) The algorithm treats this case as reading a value different from the expected one, and so t_2 completes the MwCAS operation with an unsuccessful status. We modified the code that installs word descriptors to retry if a dirty value is encountered, and this revision eliminated the linearizability anomaly in our tests.

Our case study demonstrates the power of the analyzer to detect previously-unknown bugs on codebases. Once instrumented, we can run and analyze tests with different initial configuration parameters such as number of threads, number of words accessed by each operation, and address space size. The analyzer elucidates subtle issues, reducing the time required to analyze code manually. To conclude this section, we point out that the bug we detected in *CodebaseA* is related directly to the mechanism used to ensure persistence of the data, namely the dirty bit, and likely would not exist in an atomic multi-word swap implementation designed for conventional volatile memory. Somewhat surprisingly, we were able to catch such a bug without considering crash failures.

7 Related Work

The problems of defining and analyzing linearizability can be traced back to the seminal papers of Lamport and Misra, who formalized the correct behavior of read/write register objects under concurrent access. Lamport [22] introduced safe, regular, and atomic registers in a model where processes may apply read and write operations concurrently, but writes are sequential, and every operation has both an invocation and a response. Misra's Axioms [26] accommodate multi-writer registers, and assume that "all values written by write operations are distinct," which is a natural way to establish a "reads-from" mapping. Herlihy and Wing's linearizability property [19] generalizes Lamport's atomic register in a number of ways: it covers arbitrary typed shared objects; it does not impose restrictions on concurrency (e.g., among writers); and it accommodates pending operations, which lack response events. Linearizability is the gold standard for correctness of shared objects, and is compositional in the following sense: a history involving multiple shared objects is linearizable if and only if all the maximal single-object subhistories are individually linearizable. This property is called *locality*, and can be exploited to parallelize a linearizability analyzer. Horn and Kroening generalized the idea behind locality to operations on the same object, and defined a more fine-grained composition property called *P-compositionality* [20].

Growing interest in implementing shared objects using persistent memory has exposed an important limitation of linearizability: it does not define correct behavior in the case when an operation is interrupted by a failure before it produces a response, and its caller then recovers to perform additional operations on the same object. Several extensions have been proposed to linearizability that address precisely this point, including Aguilera and Frølund's *strict linearizability* [2], Guerraoui and Levy's *persistent atomicity* [17], Berryhill, Golab and Tripunitara's *recoverable linearizability*, Izraelevitz, Mendes and Scott's *durable linearizability* [21], as well as Attiya, Ben-Baruch and Hendler's *nesting-safe recoverable linearizability* [4]. All of these properties are compatible with *detectability* [14], which is the ability to determine the outcome of an operation interrupted by a crash during subsequent recovery.

The problem of deciding whether a history of invocation and response steps satisfies a given consistency property has been studied widely in the context of ordinary linearizability. Gibbons and Korach [15] proved that deciding linearizability is NP-complete for read/write registers. They also introduced the efficient zone-based approach (see Section 4) for the special case when the reads-from mapping is known, including for histories that contain single-word Read-Modify-Write operations (e.g., successful CAS or unconditional swaps) in addition to reads and writes. On the other hand, the graph-based approach is rooted in the theory of database concurrency control [6], and has been used in several studies of consistency in distributed read/write (i.e., key-value) storage systems [3, 5, 16].

Deciding linearizability for types other than read/write registers is a challenging research problem. Automated model checking techniques based on exhaustive state space exploration [1, 9, 10, 25, 28, 29] can accommodate arbitrary data types but are limited to small inputs due to the state space explosion problem. Several more efficient techniques have been devised in the context of collection types (e.g., queue, stack, map, set). Efficient reductions from deciding linearizability to known problems have been proposed by Emmi and Enea [11] (to Horn satisfiability), and by Bouajjani et al. [8] (to control-state reachability). Ozkan, Majumdar and Niksic [27] proved that most histories over collection types can be analyzed efficiently using *hitting families*. Feldman et al. [13] proposed *local view arguments* to simplify linearizability proofs for search trees and skip lists. Our techniques are most similar to [8, 11] in that we decide linearizability automatically in polynomial time, but we focus on a different category of object types and we solve the problem directly rather than by a reduction.

8 Conclusion

This paper described an efficient graph-theoretic technique for deciding linearizability over histories of multi-word read and Compare-And-Swap operations. In our case study, the technique required only small modifications to the synchronization primitive's implementation to capture additional detail regarding unsuccessful Compare-And-Swap operations, and was shown to scale well with the size of the input history. In future work, we plan to extend our results by collecting and analyzing additional histories that include both simulated and real crash failures on a multiprocessor equipped with persistent memory.

References

1. Kiran Adhikari, James Street, Chao Wang, Yang Liu, and Shao Jie Zhang. Verifying a quantitative relaxation of linearizability via refinement. *STTT*, 18(4):393–407, 2016.
2. Marcos Aguilera and Svend Frølund. Strict linearizability and the power of aborting. *HP Labs Tech. Rep. HPL-2003-241*, 2003.
3. Eric Anderson, Xiaozhou Li, Mehul A. Shah, Joseph Tucek, and Jay J. Wylie. What Consistency Does Your Key-Value Store Actually Provide? In *Proc. of the Sixth Workshop on Hot Topics in System Dependability (HotDep)*, 2010.
4. Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. Nesting-Safe Recoverable Linearizability: Modular Constructions for Non-Volatile Memory. In *Proc. of the 37th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 7–16, 2018.
5. Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Probabilistically Bounded Staleness for Practical Partial Quorums. *PVLDB*, 5(8):776–787, 2012.
6. Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

7 Ryan Berryhill, Wojciech M. Golab, and Mahesh Tripunitara. Robust Shared Objects for Non-Volatile Main Memory. In *Proc. of the 19th International Conference on Principles of Distributed Systems (OPODIS)*, pages 20:1–20:17, 2015.

8 Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. On Reducing Linearizability to State Reachability. *Inf. Comput.*, 261(Part 2):383–400, 2018.

9 Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. Line-up: a complete and automatic linearizability checker. In *Proc. of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 330–340, 2010.

10 Pavol Cerný, Arjun Radhakrishna, Damien Zufferey, Swarat Chaudhuri, and Rajeev Alur. Model Checking of Linearizability of Concurrent List Implementations. In *Proc. of the 22nd International Conference on Computer Aided Verification (CAV)*, pages 465–479, 2010.

11 Michael Emmi and Constantin Enea. Sound, Complete, and Tractable Linearizability Monitoring for Concurrent Collections. *Proc. ACM Program. Lang.*, 2(POPL):25:1–25:27, 2017.

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

13 Yotam M. Y. Feldman, Constantin Enea, Adam Morrison, Noam Rinetzky, and Sharon Shoham. Order out of Chaos: Proving Linearizability Using Local Views. In *Proc. of the 32nd International Symposium on Distributed Computing (DISC)*, pages 23:1–23:21, 2018.

14 Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A Persistent Lock-free Queue for Non-volatile Memory. In *Proc. of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 28–40, 2018.

15 Phillip B. Gibbons and Ephraim Korach. Testing Shared Memories. *SIAM J. Comput.*, 26(4):1208–1244, 1997.

16 Wojciech Golab, Xiaozhou Li, and Mehul A. Shah. Analyzing consistency properties for fun and profit. In *Proc. ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 197–206, 2011.

17 Rachid Guerraoui and Ron R Levy. Robust emulations of shared memory in a crash-recovery model. In *Proc. of the 24th International Conference on Distributed Computing Systems (DISC)*, pages 400–407, 2004.

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

19 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

20 Alex Horn and Daniel Kroening. Faster Linearizability Checking via P-Compositionality. In *Proc. of the International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE)*, pages 50–65, 2015.

21 Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Proc. of the 30th International Symposium on Distributed Computing (DISC)*, pages 313–327, 2016.

22 Leslie Lamport. On Interprocess Communication, Part I: Basic Formalism and Part II: Algorithms. *Distributed Computing*, 1(2):77–101, 1986.

23 Leslie Lamport. *Specifying systems: the TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

24 Leslie Lamport. The PlusCal Algorithm Language. In *Proc. of the 6th International Colloquium on Theoretical Aspects of Computing (ICTAC)*, pages 36–60, 2009.

25 Gavin Lowe. Testing for linearizability. *Concurrency and Computation: Practice and Experience*, 29(4), 2017.

26 J. Misra. Axioms for Memory Access in Asynchronous Hardware Systems. *ACM Trans. Program. Lang. Syst.*, 8(1):142–153, 1986.

27 Burcu Kulahcioglu Ozkan, Rupak Majumdar, and Filip Niksic. Checking Linearizability Using Hitting Families. In *Proc. of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 366–377, 2019.

28 Viktor Vafeiadis. Automatically Proving Linearizability. In *Proc. of the 22nd International Conference on Computer Aided Verification (CAV)*, pages 450–464, 2010.

29 Martin T. Vechev, Eran Yahav, and Greta Yorsh. Experience with Model Checking Linearizability. In *Proc. of the 16th International Workshop on Model Checking Software (SPIN)*, pages 261–278, 2009.

30 Tianzheng Wang, Justin J. Levandoski, and Per-Åke Larson. Easy Lock-Free Indexing in Non-Volatile Memory. In *Proc. of the 34th IEEE International Conference on Data Engineering (ICDE)*, pages 461–472, 2018.

Consensus in Equilibrium: Can One Against All Decide Fairly?

Itay Harel
Tel-Aviv University, Israel

Amit Jacob-Fanani
Tel-Aviv University, Israel

Moshe Sulamy
Tel-Aviv University, Israel

Yehuda Afek
Tel-Aviv University, Israel

Abstract

Is there an equilibrium for distributed consensus when all agents except one collude to steer the decision value towards their preference? If an equilibrium exists, then an $n-1$ size coalition cannot do better by deviating from the algorithm, even if it prefers a different decision value. We show that an equilibrium exists under this condition only if the number of agents in the network is odd and the decision is binary (among two possible input values). That is, in this framework we provide a separation between binary and multi-valued consensus. Moreover, the input and output distribution must be uniform, regardless of the communication model (synchronous or asynchronous). Furthermore, we define a new problem - Resilient Input Sharing (RIS), and use it to find an *iff* condition for the $(n-1)$-resilient equilibrium for deterministic binary consensus, essentially showing that an equilibrium for deterministic consensus is equivalent to each agent learning all the other inputs in some strong sense. Finally, we note that $(n-2)$-resilient equilibrium for binary consensus is possible for any n. The case of $(n-2)$-resilient equilibrium for *multi-valued* consensus is left open.

2012 ACM Subject Classification Theory of computation → Distributed computing models

Keywords and phrases distributed computing, game theory, rational agents, consensus

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.20

Funding This work is partially supported by a grant from the Blavatnik Cyber Security Council, the Blavatnik Family Computer Science Research Fund and the KIPP Foundation

1 Introduction

In recent years, there is a growing interest in distributed algorithms for networks of rational agents that may deviate from the prescribed algorithm in order to increase their profit [1, 2, 3, 6, 13]. For example, an agent may have a higher profit if zero is decided in a consensus algorithm, or an agent may prefer to be (or not to be) the elected leader in a leader election algorithm. The goal is to design distributed algorithms that reach *equilibrium*, that is, where no agent can profit by cheating.

In this paper we study the consensus problem in a network of rational agents, in which each agent has a preferred decision value. We consider $(n-1)$-resilient equilibrium, that is, an equilibrium that is resilient to any coalition of up to $n-1$ agents that may collude in order to increase their expected profit (utility). This problem was proposed in [3] and studied also in [4], where the authors suggest an $(n-1)$-resilient equilibrium for binary consensus in a synchronous ring.

© Itay Harel, Amit Jacob-Fanani, Moshe Sulamy, and Yehuda Afek;
licensed under Creative Commons License CC-BY
23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller; Article No. 20; pp. 20:1–20:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

We prove that in any $(n-1)$-resilient equilibrium for binary consensus, the output of the agents must be the XOR of the inputs of all agents. Thus, due to validity, there is *no* $(n-1)$-resilient equilibrium for binary consensus in *even* sized networks, and the algorithm in [4] works well only for odd sized networks. Still, we show that the algorithm in [4] reaches $(n-2)$-resilient equilibrium for binary consensus with uniform input distribution, for *any* n.

We further show that multi-valued consensus is impossible, i.e., there is no $(n-1)$-resilient equilibrium for multi-valued consensus for $r > 2$ where r is the number of possible values, thus surprisingly there is a computational gap between binary and multi-valued consensus in this model. Note that it was previously shown that in this game theoretic model, leader election is also not equivalent to consensus [4].

Furthermore, we show that in this model, *deterministic* binary consensus is equivalent to resilient input sharing (RIS), a natural problem in distributed computing in which each agent i shares its input with all other agents in the network (a variant of the knowledge sharing problem defined in [4]). That is, in any odd sized network with uniform input distribution, any algorithm for RIS can be transformed into a $(n-1)$-resilient equilibrium for deterministic binary consensus and vice versa. Thus, providing a sufficient and necessary condition for $(n-1)$-resilient equilibrium for deterministic binary consensus.

1.1 Our Contributions

are as follows:

(§3.1) Any $(n-1)$-resilient equilibrium for binary consensus decides on the XOR of all input values.

(§3.2) In any $(n-1)$-resilient equilibrium for binary consensus the input and output distributions are uniform.

(§3.2.1) The protocol suggested in [4] reaches $(n-2)$-resilient equilibrium for binary consensus with uniform input distribution, for any n.

(§4) There is no $(n-1)$-resilient equilibrium for multi-valued consensus for $r > 2$ possible inputs.

(§5) *Deterministic* $(n-1)$-resilient equilibrium for binary consensus in a network exists *iff*:

1. The network size is odd.
2. The input distribution is uniform.
3. An equilibrium for Resilient Input Sharing (RIS) is possible in the network topology.

The model, notations and some definitions are given in Section 2, and we discuss our results and further thoughts in Section 6.

1.2 Related Work

The secret sharing problem [16] initiated the connection between distributed computing and game theory. Further works in this line of research considered multiparty communication with Byzantine and rational agents [1, 8, 11, 12, 15].

In [3], the first distributed protocols for a network of rational agents are presented, specifically protocols for *fair* leader election. In [4], the authors continue this line of research by providing basic building blocks for game theoretic distributed algorithms, namely a wake-up and knowledge sharing building blocks that are in equilibrium, and equilibria for consensus, renaming, and leader election are presented using these building blocks. The consensus algorithm in [4] claims to reach $(n-1)$-resilient equilibrium in a ring or complete network, using the knowledge sharing building block to share the input of all processors in the network, and outputting the XOR of all inputs. Consensus was further researched in [14],

where the authors show that there is no ex-post Nash equilibrium for rational consensus, and present a Nash equilibrium that tolerates f failures under some minimal assumptions on the failure pattern. Equilibrium for fair leader election and fair coin toss are also presented and discussed in [17], where it is shown to be resilient only to coalitions of sub-linear size, and a modification to the leader election protocol from [3, 4] that is resilient to every coalition of size $\Theta(\sqrt{n})$ is proposed.

In [5], the authors examine the impact of a-priori knowledge of the network size on the equilibrium of distributed algorithms, assuming the *id* space is unlimited and thus vulnerable to a Sybil attack [9]. In [7] the authors remove this assumption and assume the *id* space is bounded, examining the relation between the size of the *id* space and the number of agents in the network in which an equilibrium is possible.

2 Model

We use the standard message-passing model, where the network is a bidirectional graph $G = (V, E)$ with $|V| = n$ nodes, each node representing a *rational* agent, following the model in [2, 3]. We assume n is a-priori known to all agents, G is 2-vertex-connected, and all agents start the protocol together, i.e., all agents wake-up at the same time. We can use the Wake-Up [4] building block to relax this assumption. In Sections 3 and 4 the results apply for both synchronous and asynchronous communication networks, while Section 5 assumes a synchronous network.

In the consensus problem, each agent i has an id id_i and an input $I_i \in \{0, ...r-1\}$ and must output a decision $D_i \in \{0, ...r-1, \perp\}$. The $\perp$ output can be output by an agent to abort the protocol when a deviation by another agent is detected. A protocol achieves consensus if it satisfies the following [10]:

- **Agreement**: All agents decide on the same value, $\forall i, j : D_i = D_j$.
- **Validity**: If v was decided then it was the input of some agent, $\forall j \exists i : D_j = I_i$.
- **Termination**: Every agent eventually decides, $\forall i : D_i \neq \perp$.

▶ **Definition 1** (Protocol Outcome). *The outcome of the protocol is determined by the input and output of all agents. An outcome is* legal *if it satisfies agreement, validity, and termination, otherwise the outcome is* erroneous.

Considering individual rational agents, each agent i has a utility function U_i over the possible outcomes of the protocol. The higher the value assigned by U_i to an outcome, the better this outcome is for i. We assume the utility function U_i of each agent i satisfies *Solution Preference* [3]:

▶ **Definition 2** (Solution Preference). *The utility function U_i of any agent i never assigns a higher utility to an erroneous outcome than to a legal one.*

Thus, the Solution Preference guarantees that an agent never has an incentive to sabotage the protocol, that is, to prefer an outcome that falsifies either agreement or validity, or termination. However, agents may take risks that might lead to erroneous outcomes if these risks also lead to a legal outcome which increases their expected utility, that is, if these risks increase the expected utility that the agent is expected to gain.

An intuitive example for a utility function of an agent I with a preference towards a decision value of 1 is:

$$U_i = \begin{cases} 1 & \exists j : I_j = 1 \wedge \forall k : D_k = 1 \text{ (1 is decided by all agents)} \\ 0 & \text{otherwise (0 is decided or erroneous outcome)} \end{cases}$$

All agents are given a protocol at the start of the execution, but any agent may deviate and execute a different protocol if it increases its expected utility. A protocol is said to *reach equilibrium* if no agent can unilaterally increase its expected utility by deviating from the protocol.

▶ **Definition 3** (Nash Equilibrium[1]). *A protocol Φ is said to reach equilibrium if, for any agent i, there is no protocol $\Psi \neq \Phi$ that i may execute and leads to a higher expected utility for i, assuming all other agents follow Φ.*

2.1 Coalitions

We define a coalition of size t as a set of t rational agents that cooperate to increase the utility of each agent in t. A protocol that reaches t-resilient equilibrium [3] is resilient to coalitions of size up to t, that is, no group of t agents or less has an incentive to collude and deviate from the protocol. We assume coalition members may agree on a deviation from the protocol in advance, but can communicate only over the network links during the protocol execution.

▶ **Definition 4** (t-resilient Equilibrium). *A protocol Φ is said to reach t-resilient equilibrium if, for any group of agents $C \subset V$ s.t., $|C| \leq t$, there is no protocol $\Psi(\neq \Phi)$ that agents in C may execute and which would lead to a higher expected utility for each agent in C, assuming all agents not in C follow Φ.*

The same intuitive example for a utility function above holds for a coalition, in which the coalition has a preference towards a decision value 1.

2.2 Notations

The following notations are used throughout this paper:

- S_{-i} - all possible input vectors of agents in $V \setminus \{i\}$.
- $\#(b)$ - the number of agents in V that receive b as input.
- $\#_{-i}(b)$ - the number of agents in $V \setminus \{i\}$ that receive b as input.
- I_i - the input of agent i.
- D_i - the output value decided by agent i at the end of the algorithm.
- r - the number of possible input and output values. For binary consensus: $r = 2$.

3 Necessary Conditions for $(n-1)$-resilient Consensus

▶ **Theorem 5.** *The decision of any $(n-1)$-resilient equilibrium for binary consensus must be the XOR of all inputs, that is, $\forall i : D_i = \bigoplus_{j \in V} I_j = \sum_{j \in V} I_j \mod 2$*

Before we turn to the proof of Theorem 5 given in sections 3.1 and 3.2, note that according to this theorem, if n is even and all inputs are 1 the decision must be 0, contradicting validity and leading to the following corollary:

▶ **Corollary 6.** *There is no $(n-1)$-resilient equilibrium for binary consensus for even sized networks*

[1] Previous works defined equilibrium over each step of the protocol. For convenience, this definition is slightly different, but it is easy to see that it is equivalent.

3.1 Output is the XOR of the Inputs

Here we prove Theorem 5 based on the following two theorems, that are proved in Section 3.2:

▶ **Theorem 7.** *If the distribution over the inputs is not uniform, there is no $(n-1)$-resilient equilibrium for consensus, i.e.:* $\forall v_1, v_2 : P[I_i = v_1] = P[I_i = v_2] = \frac{1}{r}$

▶ **Theorem 8.** *In any $(n-1)$-resilient equilibrium for consensus, given any $n-1$ inputs, the distribution over the possible decision values is uniform:* $\forall s \in S_{-i}, v \in \{0, ...r-1\} : P[D_i = v|s] = \frac{1}{r}$

Notice that while the proof of theorem 5 holds only for *binary* consensus, theorems 7 and 8 are correct for multi-valued consensus as well.

Proof of Theorem 5. We prove that the decision value of binary consensus must be the XOR of all inputs using induction on $\#(1)$, the number of agents in the network whose input value is 1.

In the base case $\#(1) = 0$, the input of all agents is 0. By validity the decision must be 0.

For clarity of exposition we spell out the next case of the induction, $\#(1) = 1$, i.e., the input of one agent is 1 and of all other $n-1$ agents is 0. Assume by contradiction that the probability that 0 is decided in this case is greater than 0, i.e.,

$$\exists i \in V : \; P[D_i = 0 \mid \#_{-i}(0) = 0 \;\wedge\; I_i = 1] = p > 0$$

Let s_0 be an input configuration for a coalition in which all members of the coalition (i.e., $V \setminus \{i\}$) claim to receive 0 as input, i.e., $\#_{-i}(1) = 0$. Notice that:

$$\begin{aligned} P[D_i = 0|\ s_0] &= P[I_i = 0] \cdot P[D_i = 0 \mid s_0 \wedge I_i = 0] + P[I_i = 1] \cdot P[D_i = 0 \mid s_0 \wedge I_i = 1] \\ &= P[I_i = 0] \cdot P[D_i = 0 \mid \textit{base case}\ ] + P[I_i = 1] \cdot P[D_i = 0 \mid s_0 \wedge I_i = 1] \\ &= P[I_i = 0] \cdot 1 + P[I_i = 1] \cdot p \end{aligned}$$

By Theorem 7 (and since this is binary consensus) it follows that:

$$P[D_i = 0|\ s_0] = \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot p > \frac{1}{2}$$

Thus, contradicting Theorem 8 and proving that, $\forall i \in V$: $P[D_i = 0 \mid \#_{-i}(0) = 0 \wedge I_i = 1] = 0$ Thus if $\#(1) = 1$, the decision value must be 1, proving the first induction step.

By the inductive assumption, $\forall \#(1) < m$ the decision value of the consensus must be the XOR of all inputs, i.e., $\#(1)\ mod\ 2$. Let s_{m-1} be an input configuration for the coalition $(V \setminus \{i\})$ in which $\#_{-i}(1) = m-1$, that is, $m-1$ members of the coalition claim to receive 1, and the rest 0.

From Theorem 8 (and since this is binary consensus) we get:

$$P[D_i = (m\ mod\ 2) \mid s_{m-1}] = \frac{1}{2}$$

If $I_i = 0$ (which from Theorem 7 happens with probability $\frac{1}{2}$) and the coalition acts as if its input is s_{m-1}, then $\#(1) = m-1$. By the induction hypothesis, in such a case the decision value of the consensus must be $m-1\ mod\ 2$. To satisfy the equation above it must hold that:

$$P[D_i = (m\ mod\ 2) \mid s_{m-1} \wedge I_i = 1] = 1$$

Hence, in case $\#(1) = m$, the decision value must be $m\ mod\ 2$ - the XOR of all inputs. ◀

3.2 Proving Theorems 7 and 8

While the above proof holds only for *binary* consensus, the following lemmas and theorems are correct for multi-valued consensus.

▶ **Lemma 9.** *In any $(n-1)$-resilient equilibrium for consensus, for any $v \in \{0, \dots, r-1\}$, given any $n-1$ inputs, the probability to decide v is the same:*

$$\forall i \in V, s_1, s_2 \in S_{-i}, v : P[D_i = v|s_1] = P[D_i = v|s_2]$$

Proof. Assume by contradiction that $\exists i \in V, s_1, s_2 \in S_{-i}, v : P[D_i = v|s_1] < P[D_i = v|s_2]$. A coalition $C = V \setminus \{i\}$ with a preference to decide v, and that receives s_1 as input, has an incentive to deviate and act as if their input is s_2, contradicting equilibrium. ◀

▶ **Lemma 10.** *In any $(n-1)$-resilient equilibrium for consensus, for any input $v \in \{0, \dots, r-1\}$, the probability to decide v is the same as the probability to receive v as an input:*

$$\forall i \in V, s \in S_{-i}, v : P[D_i = v|s] = P[I_i = v]$$

Proof. For any v, if all inputs are v then by validity v is decided. For any agent i, let $\tilde{s} = (v, \dots, v) \in S_{-i}$, then due to validity, the probability that v is decided is at least $P[I_i = v]$, i.e., $P[D_i = v|\tilde{s}] \geq P[I_i = v]$. By Lemma 9 this is true for any $s \in S_{-i}$. Thus, $P[D_i = v] \geq P[I_i = v]$. Since $\sum_v P[D_i = v] = 1$ and $\sum_v P[I_i = v] = 1$, then: $\forall s \in S_{-i}, v : P[D_i = v|s] = P[I_i = v]$. ◀

Proof of Theorem 7. Assume by contradiction that $\exists v_1, v_2 : P[I_i = v_1] > P[I_i = v_2]$.

If all agents receive as input the same value v_1, then by validity v_1 is decided. Given $s = (v_1, \dots, v_1) \in S_{-i}$, the probability that v_1 is decided is at least the probability that the input of agent i is v_1, i.e., $P[D_i = v_1|s] \geq P[I_i = v_1]$.

If $n-1$ agents receive v_1 as input and one agent receives $v_2 \neq v_1$ as input the decision must not be v_1 otherwise $P[D_i = v_1|s] > P[I_i = v_1]$ contradicting Lemma 10, thus due to validity the decision must be v_2 when $n-1$ agents receive v_1 and one agent receives v_2.

Let $s' = (v_2, v_1, \dots, v_1) \in S_{-i}$. If agent i receives v_1 as input then as stated above v_2 is decided, thus: $P[D_i = v_2|s'] \geq P[I_i = v_1] > P[I_i = v_2]$, contradicting Lemma 10.

Thus, the input distribution must be uniform, i.e.: $\forall v_1, v_2 : P[I_i = v_1] = P[I_i = v_2] = \frac{1}{r}$. ◀

Proof of Theorem 8. Combining Lemma 10 with Theorem 7 :

$$\forall s \in S_{-i}, v \in \{0, ...r-1\} : P[D_i = v|s] = P[I_i = v] = \frac{1}{r}$$ ◀

3.2.1 $(n-2)$-resilient Binary Consensus for any n

A binary consensus protocol for any n is presented in [4] combining a leader election algorithm with a XOR on selected inputs. In Appendix A we prove that this protocol reaches $(n-2)$-resilient equilibrium for binary consensus for any n, when the input distribution is uniform. Note that the algorithm in [4] does not work in any network topology, but on any network in which Resilient Input Sharing is possible (see [4] and Section 5).

4 No $(n-1)$-resilient Equilibrium for Multi Valued Consensus

Here we discuss multi-valued consensus, where the agreement is between $r > 2$ possible values rather than two values. Applying the same logic as in the proof of Theorem 5 one can deduce:

▶ **Lemma 11.**
1. $\forall i \in V, v \in \{0, \ldots, r-1\} : P[D_i = v | \#(0) = n-1 \wedge \#(v) = 1] = 1$
2. $\forall i \in V, v \in \{0, \ldots, r-1\} : P[D_i = 0 | \#(0) = n-2 \wedge \#(v) = 2] = 1$

Proof. The proof is the same as the first and second induction steps in the proof of Theorem 5. ◀

▶ **Theorem 12.** *There is no $(n-1)$-resilient equilibrium for multi-valued consensus for any $r > 2$.*

Proof. Assume towards a contradiction that there is an $(n-1)$-resilient equilibrium for multi-valued consensus for some $r > 2$. Let $v, u \in \{1, \ldots, r-1\}$ s.t. $v \neq u$. Denote by X any configuration in which the input of one agent is v, of another is u, and of the rest is 0. In a run of the protocol starting from X, due to validity the network's decision value must be either 0 or u or v. We prove that none of these values can be decided in an equilibrium, reaching a contradiction. Consider some Agent i and coalition $V \setminus \{i\}$. Define s_v and s_u as follows:

- $s_v :=$ a configuration in which $\#_{-i}(0) = n-2$, $\#_{-i}(v) = 1$
- $s_u :=$ a configuration in which $\#_{-i}(0) = n-2$, $\#_{-i}(u) = 1$

Assume towards a contradiction that $P[D_i = 0 | s_v \wedge I_i = u] = p > 0$. Notice that $(s_v \wedge I_i = u) \in X$.

By point 2 of Lemma 11, if $I_i = v$ and the coalition acts as if their input vector is s_v, then i must decide 0. By Theorem 7, $P[I_i = v] = \frac{1}{r}$, therefore, $P[D_i = 0 | s_v] \geq \frac{1}{r} + \frac{p}{r} > \frac{1}{r}$, contradicting Lemma 10. Thus, in an equilibrium starting from configuration X, the decision value cannot be 0.

Assume towards a contradiction that: $P[D_i = v | s_v \wedge I_i = u] = p > 0$.

Notice that from point 1 of Lemma 11, if $I_i = 0$ and the coalition acts as if their input vector is s_v, then i must decide upon v. As before we get: $P[D_i = v | s_v] \geq \frac{1}{r} + \frac{p}{r} > \frac{1}{r}$, contradicting Lemma 10. Thus, in an equilibrium starting from configuration X, the decision value cannot be v.

Applying the symmetric claim for u, with a coalition that acts as if their input vector is s_u, we get that in an equilibrium starting from configuration X, the decision value cannot be u.

Thus, no value from $\{0, u, v\}$ can be decided in an $(n-1)$-resilient equilibrium for multi-valued consensus starting with configuration X. Hence, due to validity there is no $(n-1)$-resilient equilibrium for r-valued consensus for any $r > 2$. ◀

5 Necessary and Sufficient conditions for Deterministic Consensus

The necessary conditions from Section 3 are extended here into necessary and *sufficient* conditions for a *deterministic* $(n-1)$-resilient equilibrium for binary consensus. Deterministic means that the step of each agent in each round of the algorithm is determined completely by its input and the history of messages it has received up until the current round. In Appendix C some difficulties in trying to extend our proof to non-deterministic algorithms are provided. For the sufficient condition, a new problem - Resilient Input Sharing (RIS), a variant of knowledge sharing [4], is introduced.

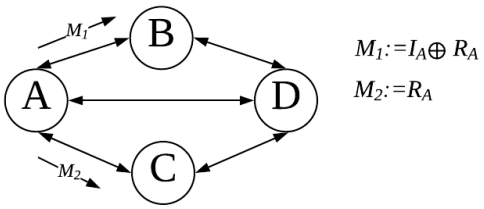


Figure 1 Messages sent by agent A at round 0. R_A is a random number chosen by A.

▶ **Theorem 13.** *A deterministic $(n-1)$-resilient equilibrium for consensus exists* iff*:*

1. *n is odd*
2. *The input distribution is uniform*
3. *There exists an algorithm for deterministic RIS (defined below).*

5.1 The Resilient Input Sharing Problem

In the RIS problem, agents in V share their binary inputs while each agent i assumes $V \setminus \{i\}$ are in a coalition. Intuitively, each agent requires all other agents to commit their inputs before or simultaneously to them learning about its input. The motivation for this requirement is that we consider problems in which (1) all agents compute the same function on the inputs, and (2) if any one input is unknown, then any output in the range of the function is still equally possible [4, 5]. Therefore the above requirement ensures that the coalition cannot affect the computation after learning the remaining (honest) agent's input, which is necessary for the computation to reach $(n-1)$-resilient equilibrium. We use the following definitions:

- K_j^t- Agent j's knowledge at the beginning of round t, including any information the coalition could have shared with it.
- Agent j is an i-knower(t)- if at the beginning of round t it can make a 'good' guess about I_i, i.e., $\exists b \in \{0,1\} : P[I_i = b | K_j^t] > P[I_i = b]$
- $Know(i,t)$ - the group of all i-knowers at the beginning of round t. In a RIS algorithm, $Know(i,0) = \varnothing$ and $Know(i,\infty) = V \setminus \{i\}$

Consider for example the network in Figure 1. At Round 0, A sends two different messages, whose XOR is its input, to B and C. At Round 1, B and C can pass these messages to D, even if this would not happen in a correct run. Thus: $Know(A,2) = \{D\}$, and $Know(A,3) = \{B,C,D\}$.

5.1.1 The RIS Problem

A solution to the RIS problem satisfies the following conditions:

1. **Termination** - the algorithm must eventually terminate.
2. **Input-sharing** - at termination, each agent knows the inputs of all other agents.
3. **Resilient** - at any round t, Agent i does not receive new information from agents in $Know(i,t)$.

Notice: in a consensus protocol, if j is an i-knower(t), and j can still influence the output at round t, then the protocol is not an $(n-1)$-resilient equilibrium. Thus, in an $(n-1)$-resilient equilibrium for consensus, no new information can be sent to i from any i-knower(t) at round t.

5.2 The effect of messages in a XOR computation

We prove that at the end of a distributed XOR computing algorithm, if an agent is given all the chains of messages that have affected its run, it can infer the input of every other agent (Theorem 19). This result applies for both deterministic and non-deterministic XOR algorithms.

▶ Remark 14. In synchronous networks, an agent can pass information to its neighbor through a silent round. Hereafter, every protocol in which informative silent rounds (explained in the proof of Lemma 20 and defined formally in Appendix B) occur is altered, and a special message $EMPTY$ is sent instead on the corresponding link.

▶ Remark 15. Hereafter, we consider networks in which every agent knows the topology of the network before the algorithm starts. Otherwise, the coalition could always cheat and choose a topology in which RIS is not possible (for example, 1-connected topology)

▶ **Definition 16** (Messages recipient). *Let R be a run of the protocol and $C \subseteq V$ a group of agents.*

$$Recv(C, t, R) = \{i \in V | i \text{ received a message from } C \text{ in round } t \text{ of } R\}$$

▶ **Definition 17** (Agents affected by a message). *In a run R, let m be a message sent at round t_m to dst_m = agent j from src_m. Then:*

- $Aff_{(m,R,t_m)} = \{j\}$ *- Agent j is directly affected by m.*
- $\forall k > 0 : Aff_{(m,R,t_m+k)} = Aff_{(m,R,t_m+k-1)} \cup Recv(Aff_{(m,R,t_m+k-1)}, R, t_m+k)$ *- Agents that were recursively affected by m.*

$Aff_{(m,R,t)}$ illustrates that a message may affect more than just its recipient; Its potential effect propagates through the network, reaching different agents through other messages.

▶ **Definition 18** (All the (chains of) messages that have an effect on agent i in run R).

- $Aff_{(i,R)} = \{ < m, t_m, src_m, dst_m >, m \text{ sent in } R \mid i \in Aff_{(m,R,T_{end})}\}$ *(R terminates at T_{end})*

▶ **Theorem 19** (The encoding of all inputs). *Let R be a run of a distributed XOR computing algorithm. Let $i, j \in V$, Agent i can compute I_j from the following information:*

1. *I_i - its input.*
2. *Decision value i.e., the XOR of all inputs.*
3. *$Aff_{(i,R)}$ - all the messages in R that have an effect on Agent i.*

To prove Theorem 19, assume the following base case is correct (to be proved in the sequel):

▶ **Lemma 20.** *Theorem 19 is correct for a network of size 3, $V = \{i, j, k\}$.*

Proof of Theorem 19. Let $G = (V, E)$ be a network where $n > 3$, such that $i, j \in V$. Create a new network G' in which agents i and j are as in G, but all other agents in $V \setminus \{i, j\}$ are clustered into one 'virtual' agent k. A distributed XOR algorithm for G' is:

- Agent k chooses $n - 2$ bits such that the XOR of these bits is its I_k.
- Agents i and j behave in G' as if they were in G, explicitly attaching to each message the id of its destination, while k emulates the behavior of the other $n - 2$ agents in V, attaching to each message the id of its source.

Let I_i^R and D_i^R be the input and output of i in run R. For any run R of the algorithm in G, $\exists R'$ - a run of the algorithm in G' s.t.,: **(1)** $I_i^R = I_i^{R'}$, $I_j^R = I_j^{R'}$, **(2)** $D_i^R = D_i^{R'}$ and **(3)** $Aff_{(i,R)} \supseteq Aff_{(i,R')}$.

From lemma 20 we know that from $D_i^{R'}$, $I_i^{R'}$ and $Aff_{(i,R')}$, I'_j can be computed. Therefore: $\forall i \neq j \in V$: - D_i^R, I_i^R and $Aff_{(i,R)}$ are enough to compute I_j^R. ◀

Proof of 20. $V = \{i, j, k\}$. Assume towards a contradiction that $\exists R_1, R_2$, two runs of the algorithm such that

1. $I_i^{R_1} = I_i^{R_2}$ - Agent i's inputs in R_1 and R_2 are the same.
2. $\bigoplus_{l \in V} I_l^{R_1} = \bigoplus_{l \in V} I_l^{R_2}$ - The decision value is the same in both R_1 and R_2.
3. $Aff_{(i,R_1)} = Aff_{(i,R_2)}$ - Exactly the same set of messages affect i in both runs.
4. $I_j^{R_1} \neq I_j^{R_2}$ - Agent j's input in R_1 is different than in R_2.

Clearly from 1, 2, and 4 it must be that $I_k^{R_1} \neq I_k^{R_2}$.

Towards a contradiction we construct run R_3, in which i's and k's inputs are the same as in R_1 and j's input is the same as in R_2, but the decision value (XOR) in R_3 is the same as in R_1.

In R_3, agents i and k start to perform their steps according to R_1 until the first round in which i or k receive a message that either does not receive in that round in R_1. Agent j behaves the same as in R_2, until the first round, denoted round $T - 1$, in which it receives a message m it does not receive in that round in R_2. Notice that it is legal for all agents to act this way in round 0. Further, if i and k can continue according to R_1 and j can continue according to R_2 until termination, then i outputs the same value as it would in R_1, which is incorrect for R_3.

Observation 1 From round T until termination j cannot send messages to i in either R_1 or R_2 or otherwise, m's effect would propagate to i, causing - $Aff_{(i,R_1)} \neq Aff_{(i,R_2)}$, contradicting point 3 of the assumptions.

Observation 2 Similarly from round T until termination, j cannot send messages to i in R_3 or otherwise, let $t \geq T$ be the first round (after T) of R_3 in which j sends a message to i. In R_1 - j does not send a message to i in round t (see Observation 1). This means that this silent round t of R_1 between j and i is informative (it tells i that the run is R_1/R_2 and not R_3). Since we do not allow informative silent rounds (see Remark 14), we reach a contradiction.

Notice that by point 3 in the assumptions, after T j cannot even communicate with i through k, since m's effect would propagate to i through k. From the two observations above, from round T of R_3, j cannot communicate with i, and from i's perspective, j is running R_1. The same logic applies for k - the first round in which it is illegal for k to act according to R_1, is a round after which k cannot send messages to i (even not through j). Thus i's experience throughout R_3 is the same as in R_1, resulting in i making an incorrect output. Contradiction. ◀

5.3 Deterministic $(n-1)$-resilient Consensus implies RIS, completing the proof

In a deterministic synchronous binary consensus protocol, in which all agents start at the same round, for each input vector the run of the algorithm is fully determined.

Let us look at a network running some deterministic binary consensus, with agent $i \in V$ and coalition $V \setminus \{i\}$. Intuitively, agents in the coalition can choose in advance an input vector to be used in the algorithm. Thus, from the coalition's perspective, there can be only two possible runs - R_0 in which $I_i = 0$, and R_1 in which $I_i = 1$. For each agent in the coalition, there is the first round in which R_0 and R_1 differ, at that point this agent knows I_i. Thus, each agent in the coalition is in one of two states - knows nothing about I_i or knows I_i, this is in contrast to non-deterministic algorithms, see for example Figure 1.

Below we transform any deterministic $(n-1)$-resilient equilibrium for binary consensus into a deterministic RIS. In Appendix C the difficulties in the non-deterministic case are explained.

▶ **Theorem 21.** *If there exists a deterministic $(n-1)$-resilient equilibrium for binary consensus, A on network $G = (V, E)$ then there exists an algorithm $\tilde{A}$ for RIS, on G.*

Proof. In $\tilde{A}$, each agent i runs A with the following modifications:

- For each message m that i receives, i appends $< m, src_m, dst_m, t_m >$ to a local buffer B of messages that has affected it.
- Agent i appends B to each message it sends.
- Agent i adds to B all the information piggy-bagged on incoming messages.

In this new algorithm $\tilde{A}$, every message propagates in the network, reaching all the agents it affects. By the end of the algorithm, the buffer maintained by agent i contains $Aff_{(i,R)}$, where R is the run of the original consensus protocol A. By theorem 5, A is a XOR computing protocol, and by theorem 19, i's buffer contains enough information to infer all inputs. Thus $\tilde{A}$ is an RIS protocol.

It remains to prove that $\tilde{A}$ is resilient. An input sharing protocol is resilient (Subsection 5.1) if at any round t, i does not receive new information from agents in $Know(i,t)$. As stated before, this demand applies for $(n-1)$-resilient equilibrium for binary consensus as well. Thus, to show that $\tilde{A}$ is resilient, it is enough to show that $\forall i \in V$:

- In each round t of $\tilde{A}$, i receives messages from the same neighbors it receives from in A
- In each round t of $\tilde{A}$, $\forall j \neq i$: $j \in Know(i,t)$ in $\tilde{A} \implies j \in Know(i,t)$ in A

The first point is immediate from the construction of $\tilde{A}$. For the second point - observe some agent j at round t of A, which is not an i-knower in A. For j to become an i-knower(t) in $\tilde{A}$, the coalition must send j enough information by t for it to make a 'good' guess about I_i. There are two kind of paths in G by which the coalition can send information to j - paths that do not pass through i, and paths that do.

Through paths not including i, the coalition can pass information in the same pace for both A and $\tilde{A}$. Since $j \notin Know(i,t)$ in A, using these paths alone is not enough to make j an i-knower(t) in $\tilde{A}$. Regarding paths that include i - as argued in the beginning of this subsection, in a *deterministic* $(n-1)$-resilient equilibrium for binary consensus, if a member of the coalition has any information about I_i, then that member *knows* I_i. Therefor, in A, i should not receive messages from members of $Know(i,t)$ at round t. Thus if the coalition has information it wants to pass to j, it cannot do so using paths including agent i, since i does not accept and propagate messages from i-knowers. To conclude, if j is an i-knower in $\tilde{A}$, j is an i-knower in A. Since A is $(n-1)$-resilient equilibrium for consensus, $\tilde{A}$ is resilient as well. ◀

5.3.1 Completing the proof, necessary and sufficient conditions for deterministic Consensus

Proof of Theorem 13⇐. Assume that the 3 conditions are realized, and let us suggest a simple $(n-1)$-resilient equilibrium for binary consensus: run the RIS algorithm and output the XOR of all inputs. Since the RIS algorithm is resilient, no coalition has an incentive to cheat. ◀

Proof of Theorem 13⇒. Assume that $(n-1)$-resilient equilibrium for binary consensus exists. By 6 and 7, n is odd and the input distribution is uniform. By theorem 21, RIS is possible. ◀

6 Discussion

Surprisingly, while there is an equilibrium for binary consensus resilient to coalitions of $n-1$ agents, no such equilibrium exists for multi valued consensus. This is the first model we know of in which there is a separation between binary and multi valued consensus. Intuitively, this is because a coalition with a preference towards v has an incentive to cheat and act as if the input of all agents in the coalition is v, thus lowering the number of possible decision values (due to validity) to two values, at most. Consider for example the standard bit-by-bit reduction from binary to multi valued consensus, the probability to decide v is now at least $\frac{1}{2}$ instead of $\frac{1}{r}$, since the decision value is determined by the decision on the first bit of the coalition input that differs from the input of the honest agent. We conjecture that this intuition holds even for smaller coalitions, up to a single cheater. The results in §3 and §4 hold regardless of the network topology, scheduling models, or cryptographic solutions, as they are based solely on the input values and utility of the agents.

Furthermore, we present necessary and sufficient conditions for $(n-1)$-resilient equilibrium for binary *deterministic* consensus using the resilient input sharing (RIS) problem. This in fact means that an agent cannot hide its input from the rest of the network in any $(n-1)$-resilient equilibrium protocol that computes XOR, i.e., even though we only compute the XOR of inputs, at the end of the protocol all agents can deduce the input values of all other agents.

There are several open directions for research:

- Extending the equivalence result to *non-deterministic* consensus and RIS.
- Can binary consensus be solved without the conditions of even size and uniform input for coalitions of a smaller size, such as $n-2$ or $\frac{n}{2}$?
- Does an equilibrium for multi-valued consensus exist for coalitions of size $n-2$ or less?

References

1. Ittai Abraham, Lorenzo Alvisi, and Joseph Y. Halpern. Distributed computing meets game theory: combining insights from two fields. *SIGACT News*, 42(2):69–76, 2011. doi:10.1145/1998037.1998055.
2. Ittai Abraham, Danny Dolev, Rica Gonen, and Joseph Y. Halpern. Distributed computing meets game theory: robust mechanisms for rational secret sharing and multiparty computation. In *PODC*, pages 53–62, 2006. doi:10.1145/1146381.1146393.
3. Ittai Abraham, Danny Dolev, and Joseph Y. Halpern. Distributed Protocols for Leader Election: A Game-Theoretic Perspective. In *DISC*, pages 61–75, 2013. doi:10.1007/978-3-642-41527-2_5.
4. Yehuda Afek, Yehonatan Ginzberg, Shir Landau Feibish, and Moshe Sulamy. Distributed Computing Building Blocks for Rational Agents. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, 2014.
5. Yehuda Afek, Shaked Rafaeli, and Moshe Sulamy. The Role of A-priori Information in Networks of Rational Agents. In Ulrich Schmid and Josef Widder, editors, *32nd International Symposium on Distributed Computing (DISC 2018)*, volume 121 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:18, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.DISC.2018.5.
6. Amitanand S. Aiyer, Lorenzo Alvisi, Allen Clement, Michael Dahlin, Jean-Philippe Martin, and Carl Porth. BAR fault tolerance for cooperative services. In *SOSP*, pages 45–58, 2005. doi:10.1145/1095810.1095816.
7. Dor Bank, Moshe Sulamy, and Eyal Waserman. Reaching Distributed Equilibrium with Limited ID Space. In *Structural Information and Communication Complexity - 25th International*

Colloquium, SIROCCO 2018, Ma'ale HaHamisha, Israel, June 18-21, 2018, Revised Selected Papers, pages 48–51, 2018. doi:10.1007/978-3-030-01325-7_9.

8 Varsha Dani, Mahnush Movahedi, Yamel Rodriguez, and Jared Saia. Scalable rational secret sharing. In *PODC*, pages 187–196, 2011. doi:10.1145/1993806.1993833.

9 John R. Douceur. The Sybil Attack. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 251–260, London, UK, UK, 2002. Springer-Verlag.

10 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32(2):374–382, April 1985. doi:10.1145/3149.214121.

11 Georg Fuchsbauer, Jonathan Katz, and David Naccache. Efficient Rational Secret Sharing in Standard Communication Networks. In *TCC*, pages 419–436, 2010. doi:10.1007/978-3-642-11799-2_25.

12 Adam Groce, Jonathan Katz, Aishwarya Thiruvengadam, and Vassilis Zikas. Byzantine Agreement with a Rational Adversary. In *ICALP (2)*, pages 561–572, 2012. doi:10.1007/978-3-642-31585-5_50.

13 Joseph Y. Halpern and Vanessa Teague. Rational secret sharing and multiparty computation: extended abstract. In *STOC*, pages 623–632, 2004. doi:10.1145/1007352.1007447.

14 Joseph Y. Halpern and Xavier Vilaça. Rational Consensus: Extended Abstract. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC '16, pages 137–146, New York, NY, USA, 2016. ACM. doi:10.1145/2933057.2933088.

15 Anna Lysyanskaya and Nikos Triandopoulos. Rationality and Adversarial Behavior in Multiparty Computation. In *CRYPTO*, pages 180–197, 2006. doi:10.1007/11818175_11.

16 Adi Shamir. How to Share a Secret. *Commun. ACM*, 22(11):612–613, 1979. doi:10.1145/359168.359176.

17 Assaf Yifrach and Yishay Mansour. Fair Leader Election for Rational Agents in Asynchronous Rings and Networks. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, PODC '18, pages 217–226, New York, NY, USA, 2018. ACM. doi:10.1145/3212734.3212767.

A $(n-2)$-resilient Consensus for Even n

In [4] the authors provide a different protocol for even and odd size networks. Here we prove that the protocol suggested for binary consensus when n is even, provides $(n-2)$-resilient equilibrium for binary consensus. The protocol assumes the existence of an $(n-2)$-resilient equilibrium for knowledge-sharing in order to perform an $(n-2)$-resilient equilibrium for leader election (notice that in [17], it is shown that in an asynchronous ring, the leader election algorithm of [4] is not resilient to coalitions of size $O(\sqrt{n})$). Further, the protocol assumes each agent has a unique id, and all agents start the protocol at the same round.

Algorithm 1 [4] Protocol for $(n-2)$-resilient equilibrium for binary consensus.

for each agent i:

1. Let $r_i = random(1, ..., n)$
2. Execute Knowledge sharing [4] to learn $K = \{< I_1, r_1, id_1 >, ... < I_n, r_n, id_n >\}$
3. For any k, if $I_k \notin \{0, 1\}$ or $r_k \notin \{1, ..., n\}$, or $\exists j$ such that $id_j = id_k$, set $D_i = \perp$ and terminate
4. Calculate $L = (\sum_{k=1}^{n} r_k) \mod n$, set *leader* to be the L-th ranked id
5. Set $D_i = \bigoplus_{\substack{k \in \{1,...,n\} \\ id_k \neq leader}} I_k$ and terminate

Essentially, the protocol suggested in [4] when n is even, performs input sharing in parallel to leader election, then outputs the XOR of all inputs without the leader's input. It is easy to see that this protocol for consensus satisfies agreement, validity, and termination.

For the rest of this section, let V be the agents in a network of even size, executing the protocol in Algorithm 1, with coalition $C = V \setminus \{i, j\}$. Also, we assume the input distribution is uniform, i.e., $\forall i \in V : P[I_i = 0] = P[I_i = 1] = \frac{1}{2}$.

▶ **Theorem 22.** *Algorithm 1 is an $(n-2)$-resilient-equilibrium for binary consensus.*

Proof of Theorem 22 follows after the following observation and lemmas:

If no agent deviates from the protocol in Algorithm 1, then the decision value of the consensus is uniformly distributed. Therefore, if after the coalition C deviates, the probability to decide a value preferred by C is still $\leq 1/2$, C has no incentive to cheat.

▶ **Lemma 23.** *If at the end of the knowledge sharing step, i learns the true value of I_j (or vice versa), C has no incentive to deviated from the protocol.*

Proof. In this case at least one of the inputs of agents i and j is not omitted from the XOR performed by i or/and j. Since the coalition has no influence on these inputs, which are uniformly distributed, the result of the XOR is also uniformly distributed, and they have no incentive to cheat. ◀

Following Observation A.0 and Lemma 23 it remains to consider the case in which the coalition can cheat each of i and j about the input, or id, and/or random value selected in step 1, of the other.

▶ **Lemma 24.** *C has no incentive to share with i and j two sets of ids and random values that disagree.*

Proof. Assume C has a preference towards v. Denote by X the case in which the coalition forced i and j to elect two different leaders. Notice that to achieve this the coalition must provide i and j two different sets of ids and random values for all the other agents.

In case X the decision value of i is independent of the decision value of j. Following [4] $\forall k, l \in V : P[leader_k = l] = \frac{1}{n}$. Thus,

$$\forall k \in \{i, j\} : \; P[leader_k \neq k] = \frac{n-1}{n}$$

If i does not elect itself as leader, then (based of the uniform input distribution) $D_i = v$ with probability $\frac{1}{2}$. Hence:

$$\begin{aligned} P[D_i = v|X] &= P[leader_i \neq i|X] \cdot \frac{1}{2} + P[leader_i = i|X] \cdot P[D_i = v|X \wedge leader_i = i] \\ &\leq P[leader_i \neq i|X] \cdot \frac{1}{2} + \frac{1}{n} \\ &= \frac{n-1}{2n} + \frac{1}{n} \\ &= \frac{n+1}{2n} \end{aligned}$$

The same goes for agent j. Since the decision of i is independent of the decision of j (by solution preference, the coalition succeeds only if $D_i = D_j = v$):

$$\forall n > 2 : P[D_i = v \wedge D_j = v|X] = P[D_i = v|X] \cdot P[D_j = v|X] \leq (\frac{n+1}{2n})^2 < \frac{1}{2}$$

Since the probability to decide v when executing the protocol in Algorithm 1 with no deviation is $\frac{1}{2}$, there is no incentive for C to share different ids or random values with i than it shares with j (and vice versa). ◀

▶ **Lemma 25.** *C has no incentive to share a set of input values with i and a set with j, that disagree.*

Proof. Assume C has a preference towards v, and denote by Y the case in which the coalition provides a set of input values with i and a set with j, that disagree. Like in the previous proof, the decision values of i and j are independent. By 24, both agents i and j elect the same leader, hence that at least one of them is not elected. *W.l.o.g*, i is not the leader. When i calculates the XOR (step 5), I_i is not omitted from the calculation. Since the set of inputs provided to i is independent for I_i (provided by knowledge-sharing being resilient), and since C does not know in advance I_i, which is uniformly distributed, the result of the XOR is uniformly distributed. I.e.: $P[D_i = v|Y] = \frac{1}{2}$. Since the probability to reach consensus on v when running Algorithm 1 with no deviation is $\frac{1}{2}$, there is no incentive for C to share different input values with i, than it shares with j. ◀

Proof of Theorem 22. From lemmas 23, 24 and 25, we know that, in any run of the algorithm, both i and j obtain the same knowledge K. Since the decision value is uniformly distributed in a correct run, then for any legal knowledge sharing K: $P[D_i = 0] = P[D_i = 1] = \frac{1}{2}$. This means that C has no incentive to choose in advance either a specific set of random values or input values or ids. ◀

B Informative Silent Rounds and Informative Messages

For this section, let R be a run of a distributed XOR algorithm A in network $G = (V, E)$.

▶ **Definition 26** (Link experiences). *For any Agent $i \in V$ at any round t, for all $(i, j) \in E$ define the incoming link experience of i to be:*

$$ILE(i, j, t) = \begin{cases} m & \text{(j sends message m to i at round t)} \\ silence & \text{(j does not send any message to i at round t)} \end{cases}$$

Similarly, define the outgoing link experience of i with j at round t to be:

$$OLE(i, j, t) = \begin{cases} m & \text{(i sends message m to j at round t)} \\ silence & \text{(i does not send any message to j at round t)} \end{cases}$$

▶ **Definition 27** (Round of an agent). *For $i \in V$ at round t:*

- *I_i := Agent i's input.*
- *$in(i, t)$:= All incoming link experiences i has with its neighbors at round t.*
- *$out(i, t)$:= All outgoing link experiences i has with its neighbors at round t.*
- *$D(i, t) \in \{0, 1, ?\}$:= The decision value of i. As long as t is not the final round, $D(i, t) =?$*
- *$round(i, t) =< I_i, in(i, t), out(i, t), D(i, t) >$:= Round t from agent i's perspective*

▶ **Definition 28** (Run of an agent). *For $i \in V$, define $R(i)$ to be the projection of R on i:*

$$R(i) :=< round(i, 0), round(i, 1), ...round(i, T_{end}) >$$

▶ **Definition 29** (Prefix and suffix of a run). *For $i \in V$:*

$$R(i)^{0\dots t} := < round(i,0), round(i,1), \dots round(i,t) >$$

I.e. the prefix of $R(i)$ up to round t. Each prefix of a run has a set of possible legal suffixes of the form:

$$S(i)^{t+1\dots} := < round(i,t+1), round(i,t+2), \dots >$$

▶ **Definition 30** (Informative link experience). *Intuitively, informative link experiences are ILE after which i's execution may be altered. Let $i, j \in V$. Denote e_1 to be a legal ILE that i has at round t of R with j. e_1 is informative if there exists:*

- *e_2 := Another ILE that i has with j at round t ($e_1 \neq e_2$)*
- *out := A set of outgoing link experiences i had with its neighbors at round t*
- *in := A set of incoming link experiences i had with its neighbors at round t* not including *j.*
- *D := A decision value.*

Such that the following holds:

1. *Both $< I_i,\ in \bigcup \{e_1\},\ out,\ D >$ and $< I_i,\ in \bigcup \{e_2\},\ out,\ D >$ are legal rounds for agent i in a run of A with prefix $R(i)^{0\dots t-1}$*
2. *$\exists\ S(i)^{t+1\dots}$ - a suffix of i's run, such that:*

$$P[S(i)^{t+1\dots} \mid R(i)^{0\dots t-1} \wedge < I_i,\ in \bigcup \{e_1\},\ out,\ D >] \neq$$
$$P[S(i)^{t+1\dots} \mid R(i)^{0\dots t-1} \wedge < I_i,\ in \bigcup \{e_2\},\ out,\ D >]$$

▶ **Definition 31** (Informative silent round). *In Subsection 5.2, an informative silent round is actually an incoming link experience i has with j at round t, such that:*

1. *$ILE(i,j,t) = silence$*
2. *$ILE(i,j,t)$ is informative*

C Difficulties in extending Theorem 21 to Non Deterministic Case

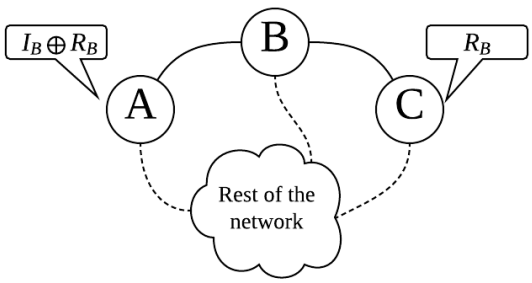


Figure 2 A snippet of agents A and C's knowledge regarding I_B in a non-deterministic XOR computing algorithm. R_B is a random number chosen by B.

Figure 2 depicts a counter example in a non-deterministic algorithm to the construction in Theorem 21. A and C cannot make a good guess regarding I_B on their own. If however, they were able to combine the information they have acquired, they would become B-knowers. In the original algorithm, B can still receive (send) messages from (to) A and C (they are not

B-knowers). Applying the construction in Theorem 21 on this non-deterministic algorithm, agent A would have been able to pass C its array of messages, and B would have to let it pass through, thus creating an A-C 'shortcut' through B.

The Evolutionary Price of Anarchy: Locally Bounded Agents in a Dynamic Virus Game

Laura Schmid
IST Austria, Klosterneuburg, Austria
lschmid@ist.ac.at

Krishnendu Chatterjee
IST Austria, Klosterneuburg, Austria
krishnendu.chatterjee@ist.ac.at

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

Abstract

The Price of Anarchy (PoA) is a well-established game-theoretic concept to shed light on coordination issues arising in open distributed systems. Leaving agents to selfishly optimize comes with the risk of ending up in sub-optimal states (in terms of performance and/or costs), compared to a centralized system design. However, the PoA relies on strong assumptions about agents' rationality (e.g., resources and information) and interactions, whereas in many distributed systems agents interact locally with bounded resources. They do so repeatedly over time (in contrast to "one-shot games"), and their strategies may evolve.

Using a more realistic evolutionary game model, this paper introduces a realized *evolutionary Price of Anarchy (ePoA)*. The ePoA allows an exploration of equilibrium selection in dynamic distributed systems with multiple equilibria, based on *local* interactions of simple *memoryless* agents.

Considering a fundamental game related to virus propagation on networks, we present analytical bounds on the ePoA in basic network topologies and for different strategy update dynamics. In particular, deriving stationary distributions of the stochastic evolutionary process, we find that the Nash equilibria are not always the most abundant states, and that different processes can feature significant off-equilibrium behavior, leading to a significantly higher ePoA compared to the PoA studied traditionally in the literature.

2012 ACM Subject Classification Theory of computation → Solution concepts in game theory; Theory of computation → Network games; Networks → Network algorithms

Keywords and phrases Evolutionary Games, Virus Propagation, Price of Anarchy, Analysis

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.21

Related Version [22], `https://arxiv.org/abs/1906.00110`

Funding *Krishnendu Chatterjee*: Austrian Science Fund (FWF) NFN Grant S11407-N23 RiSE/SHINE

1 Introduction

The performance and efficiency of large distributed systems, such as open peer-to-peer networks which any user can join, often critically depend on cooperation and can suffer if users behave selfishly, e.g.: consume but not contribute resources [16], choose routes [21] and neighbors [7] to optimize their *personal* benefits, etc. Non-cooperative behavior can also affect security. For example, if security mechanisms such as the installation of anti-virus software are employed just for self-protection, a virus may spread further than in cooperative environments [8], while at the same time increasing global security investment costs [2].

© Laura Schmid, Krishnendu Chatterjee, and Stefan Schmid;
licensed under Creative Commons License CC-BY
23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller; Article No. 21; pp. 21:1–21:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The *Price of Anarchy (PoA)* [14] is a game-theoretic concept which allows to assess to which extent a distributed system is affected negatively by non-cooperative behavior. Essentially, the PoA compares the optimal social welfare resp. cost to the welfare resp. cost in the worst Nash equilibrium: an equilibrium in which no selfish agent, henceforth called *player*, has an incentive to change its behavior. If a distributed system has a large PoA, this means that the performance resp. cost can be far from optimal: the system may require a redesign or at least strong incentive mechanisms.

However, Nash equilibria are often not a good model for real-world distributed systems, for several reasons. In particular:

1. ***Dynamic character:*** Distributed systems typically are not based on "one-shot games" but rely on dynamic interactions over time: e.g., peers (i.e., *players*) in peer-to-peer systems such as BitTorrent interact repeatedly, for example using tit-for-tat strategies, leading to *repeated games* [29].
2. ***Local information:*** Players in distributed systems often only have local information about the network, based on interactions (e.g., with neighboring players). Obtaining global information is typically infeasible, especially in large-scale distributed systems.
3. ***Bounded resources:*** Players typically also have only limited resources, e.g., in terms of memory or in terms of the complexity of the kind of algorithms they can execute.

This paper is motivated by the desire to extend the concept of Price of Anarchy to account for these important characteristics of distributed systems. While the research community is currently discussing alternatives to Nash equilibria such as Bayes-Nash equilibria [23] for games with incomplete information, it is believed that such extensions are complex due to having to take into account players' belief systems, amongst other issues, and thus introduce major research challenges [20]. In contrast, we aim to port the PoA to *evolutionary games* in purely *local information* scenarios: games embedded in dynamical processes, where simple, *memoryless* players interact repeatedly and locally, and can update their strategies over time. This allows understanding a system's behavior over time, and helps answer questions about equilibrium selection, convergence, and possible off-equilibrium behavior.

However, analyzing equilibrium selection in stochastic processes described by evolutionary games is already challenging in 2-player games in a population with m pure strategies [10, 9]. Games *on graphs*, while also highly useful in verification and synthesis of (possibly distributed [19]) reactive systems [5, 17], are often particularly difficult, also when considering evolutionary games [1], due to additional dependencies on the possible interactions.

Our contributions. This paper extends the notion of price of anarchy to evolutionary games, introducing the *evolutionary Price of Anarchy (ePoA)*. In particular, we are interested in the equilibrium behavior of simple memoryless players, which repeatedly and *locally* interact on a graph. For this setting, the ePoA is a more natural measure of efficiency than the static PoA: it allows considering agents that don't have perfect information when making decisions, and don't necessarily always reach equilibria in the game they are playing. We note that the ePoA is essentially a framework; it can be used to study games under different evolutionary dynamics, with different parameters and different ways in which players evolve their strategies.

To shed light on how the evolutionary perspective can affect the conclusion on the possible impact of selfish behavior in distributed systems, we consider a concrete case study: the well-known virus propagation game introduced by Aspnes et al. [2]. We present an analysis of the evolutionary dynamics of this game for the three fundamental dynamic models

(related to genetic evolution and imitation dynamics) and different basic network topologies. Interestingly, while the analysis of such evolutionary games is challenging in general, we are able to provide an exact characterization of the long-run frequencies of configurations for these scenarios.

We make several interesting observations. We find that the evolutionary dynamics of this game give rise to a rich variety of behaviors. In particular, the ePoA can be significantly worse than the classic PoA, for reasonable (i.e. not too high) mutation rates. We also find that Nash equilibria are not always the most frequent, i.e., *abundant*, states, and different processes can feature significant off-equilibrium behavior.

Our analytical results are complemented with simulations, also of more complicated topologies. For the full version of the paper, including technical details, as well as a more detailed discussion of related work, see [22]. Full results and our implementation are available upon request.

2 Preliminaries

Before launching into a full description and analysis of our model, we first revisit the virus inoculation game which will serve us as a case study in this paper. We also give a short introduction to evolutionary dynamics, evolutionary games and evolutionary graph theory.

2.1 The virus inoculation game

In the classic one-shot virus inoculation game [2], nodes must choose between installing anti-virus software (inoculating themselves) at a cost, or risk infection by a virus that spreads from a random location and can reach a node if there is a path of not inoculated nodes in between. The network is modeled by an undirected graph $G = (V, E)$ with N nodes. Nodes correspond to players in the game. Every node is equipped with a strategy a_i that denotes its propensity to inoculate itself. If a_i is 0 or 1, the strategy is called *pure*. Every node takes an action according to its strategy; the overall configuration of nodes is reflected in the strategy profile $\vec{a} \in [0, 1]^n$. Inoculation costs V. After everyone has made their choice, the adversary picks a random node as the starting point of infection. The virus propagates through the network, infecting all non-inoculated nodes that have a direct path to an infected node. That is, propagation happens on an "attack graph" $G_{\vec{a}}$, where inoculated nodes have been removed, and only insecure nodes remain. Being infected then comes with a cost $I > V$. Inoculation prevents infection as well as further virus transmission by the inoculated node.

The cost of a mixed strategy for a node i in this model is therefore

$$\text{cost}_i(\vec{a}) = a_i V + (1 - a_i) I \cdot p_i(\vec{a}), \tag{1}$$

where $p_i(\vec{a})$ is the probability of node i being infected given the strategy profile $\vec{a}$ and the condition that i did not inoculate. The goal of each player is to minimize its own cost, while it does not take the resulting social cost to the system in total into account. This social cost is simply

$$\text{cost}(\vec{a}) = \sum_{j=0}^{N-1} \text{cost}_j(\vec{a}). \tag{2}$$

Aspnes et al. then showed the following characterization of pure equilibria (for the proof and the extension to mixed equilibria, cf. [2]):

▶ **Corollary 1** (Characterization of pure equilibria). *Fix V and I, and let the threshold be $t = VN/I$. A strategy profile $\vec{a}$ is a pure Nash equilibrium if and only if:*

(a) *Every component in the attack graph $G_{\vec{a}}$ has at most size t.*

(b) *Inserting any secure node j and its edges into $G_{\vec{a}}$ yields a component of size at least t.*

2.2 Evolutionary dynamics and games

Game theory considers individuals that consciously aim to reach the best outcome for them in a strategic decision situation. Its classic framework, using concepts like Nash equilibria usually makes some key assumptions about the players' rationality, their beliefs and cognitive abilities. In contrast, *evolutionary game theory*, as a generic approach to evolutionary dynamics [24, 25, 13], considers a population of players with bounded rationality instead. Each player adopts a strategy to interact with other population members in a game. The players' payoffs from these interactions – which depend on the actions of the co-players and therefore on the abundance of different strategies – are considered to be their *evolutionary fitness*. Success in the game is translated into reproductive success: good strategies reproduce faster, whereas disadvantageous strategies go extinct. In a nutshell, evolutionary game theory describes dynamics that are dependent on the frequency of different strategies in a population. The evolutionary dynamics then depends on the setup and structure of the population, the underlying game, and the way strategies spread. *Evolutionary graph theory* considers different population structures and how the dynamics are changed by changing the underlying graph [15].

3 The Evolutionary Price of Anarchy

It may not be realistic to assume that nodes in a game such as the virus game in [2] will have perfect information in the first place. In large networks, it is highly unlikely that nodes separated by many neighbors would know G and each others' decision and would then optimally react to this immense amount of information. Rather, it is more natural to think that nodes only have *local* information at most. They can see their neighbors' most recent choices and react to them only when updating their strategy, while being unaware of the choices of nodes with a higher degree of separation. Thus, our model does not use the assumption of nodes having full information or memory; at most, they need to compare their payoff with the payoffs of their neighbors. To model this, we first introduce an evolutionary virus inoculation game and three different kinds of stochastic evolutionary dynamics. We then define a general notion of the evolutionary price of anarchy.

3.1 The evolutionary virus inoculation game

We consider an evolutionary process on a static graph $G = (V, E)$ with V the set of vertices (players) and E the set of edges. The $N = |V|$ vertices are occupied by players in the game, and we say that two nodes/players are *neighbors* if there is an edge connecting them. One iteration of the evolutionary process consists of three stages:

1. *Decision making.* All players make a decision whether to inoculate themselves against possible virus infections in the network. In case they choose to do so, they pay a cost V, and pay nothing otherwise. Players' propensity to inoculate is encoded in their strategy a_i.
2. *Virus propagation.* After everyone has concurrently made a decision, expected costs of the nodes when the system is attacked by a virus are calculated. To do so, we use a process with n steps: in each step, the virus starts at a different node of the graph and

spreads throughout the network. Inoculated players are unaffected by the virus and cannot transmit it to their neighbors, while unprotected players pay a cost once they become infected, and spread the virus to their other insecure neighbors. Uninfected players who are not inoculated do not pay anything. We will use the term "realized cost vector" to describe the vector $ct = [I, V, L = 0]$, where infected nodes pay I, inoculated nodes pay V, and insecure but uninfected nodes pay nothing. Once the virus has swept over the system, infecting unprotected players and their unprotected neighbors, costs are recorded, infection status is reset, and the next virus starts at the next node, until every node has served as a starting point. Alternatively, the process can be randomized by letting sufficiently many viruses start at random nodes. Once this has happened, cumulative costs are averaged, giving expected negative payoffs for the current strategy configuration, and the next stage commences.

3. *Evolution of strategies.* After receiving their expected negative payoff, players can assess the damage done and subsequently change their strategy based on comparisons with their neighbors' payoffs, before they again decide which action to take, and the next game begins. It is important to realize that this updating process is based on purely local information: nodes only need to know their neighbors' payoffs in order to make their decisions. This means that also the outcomes that can be realized may differ from the Nash equilibria that are found in the perfect information model of Aspnes et al. We consider dynamics that can take both *selection* and *mutation* into account: strategies typically evolve according to their (frequency-dependent) fitness, but they can also randomly change with a small probability μ, which we will refer to as the *mutation rate*. This prevents absorbing states in the process and lets us compute a *unique* stationary distribution that gives the long-term frequencies of system configurations. In the limiting case of $\mu \to 0$, the process always terminates and reaches a state where either all nodes are inoculated, or none are.

We differentiate here between two kinds of well known memoryless evolutionary dynamics, but are not restricted by them; in general we can configure our framework for general monotone imitation dynamics as described in [9]:

(a) **Genetic evolution:** On one hand, we consider genetic evolution as described by the *Moran process*. In this context, we analyze two different variants: a *death-birth (DB)* and *birth-death (BD)* scenario, respectively (cf [26] and [22]). In the DB scenario, a node is picked to die in each time step of the process. The vacancy is then filled by a copy of one of its neighbors, with the probability of one node being the replacement in some (possibly non-linear) way proportional to its payoff, such that nodes with higher payoffs (or rather, fewer losses) have a higher chance of being chosen as the replacement. In the BD scenario, meanwhile, first a node is picked for reproduction in each round with probability proportional to its payoff. This node subsequently chooses one of its neighbors uniformly at random and replaces it with a copy of itself. After every update, payoffs are recomputed. To visualize an example of such a process, we illustrate the DB scenario in Fig. 1.

(b) **Cultural evolution:** On the other hand, we also consider "cultural" evolution through imitation dynamics in the form of a standard pairwise comparison process ([27] and [22]). In this well established model of evolution, a focal player picks a neighboring "role model" node uniformly at random in every time step, observes its payoff, and switches to its strategy with probability

$$\varrho = \frac{1}{1 + e^{-\beta(\pi' - \pi)}} \tag{3}$$

where π' is the payoff of the neighbor and π the node's own payoff. This function is parameterized by the *selection strength* $\beta \geq 0$, which is a measure for the noise in the update probability, and with it, how much the payoff difference influences the dynamics. Thus, for $\beta = 0$, updating is random with probability $\varrho = 1/2$, whereas for $\beta > 0$, strategies with lower costs are preferred.

These processes are simulated until *timeout*: once there have been k update steps (where k is large to ensure convergence), we calculate the average welfare of the population (which is the average sum of payoffs), as well as the average count of how often the system visited the different states, and return.

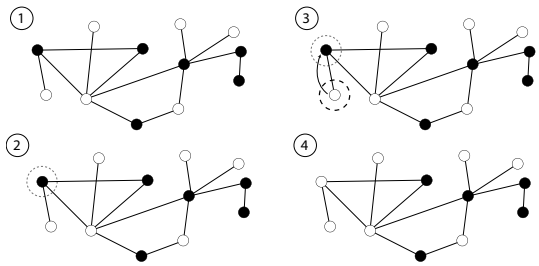


Figure 1 We illustrate the evolutionary dynamics given by the Moran Death-Birth process. Step 1: The nodes in the network shown use either Strategy 1 (white) or Strategy 2 (black). Step 2: One random node – in our example, a black one – is picked for death (visualized by the grey dashed line). Step 3: One of the nodes that neighbor the new vacancy is picked for reproduction, depending on its payoff. Here, this is a white node. Step 4: The reproducing node has passed on its strategy, such that there is one more white node on the graph.

3.2 The evolutionary price of anarchy

In the analysis of an evolutionary game, a fundamental question of interest is to which distribution different dynamics converge to, for various topologies and parameters. Such a stationary distribution of the Markov chain underlying the evolutionary dynamics contains the long-run frequencies of states. This corresponds to the long-run probabilities of finding the system in the different states. Since we obtain an ergodic process on the space of all possible population compositions (hereon called configurations) by our elementary updating rules, this limiting distribution of inoculation states exists and is unique. It is also called the *selection-mutation equilibrium* $\mathbf{x}$ of a given evolutionary process, and forms the basis of further analysis - we emphasize here that our results are therefore based on an actual limit distribution and not an arbitrary cutoff time. We note that it is the nonzero mutation rate $\mu > 0$ that provides the necessary ergodicity for a unique stationary distribution to exist – otherwise, the Markov chain would have absorbing states where all nodes are inoculated or none are.

We can subsequently find the average social cost $\hat{S}$ for any of the dynamics we use, either by averaging over the total cost in each round (when the process is simulated), or multiplying $\mathbf{x}$ with the vector $\mathbf{R}$ containing the cost of all possible configurations, such that

$$\hat{S} = \mathbf{x} \cdot \mathbf{R} = \sum_i x_i \, R_i, \tag{4}$$

where $R_i = \sum_{j=1}^{N} \hat{\pi}_j^i$, and $\hat{\pi}_j^i$ is the average payoff of player j in configuration i. We measure the efficiency of a process by comparing the average social cost $\hat{S}$ with the optimum Ω. At this point, we introduce the concept of the *evolutionary price of anarchy ePoA* as the ratio

of the average social cost (or payoffs) of a process against the social optimum. Similarly to the static PoA, we hence define, for a particular evolutionary dynamics and assuming negative payoffs:

$$ePoA = \hat{S}/\Omega. \tag{5}$$

For positive payoffs, we define $ePoA = \Omega/\hat{S}$ and note that in both cases, $ePoA \geq 1$, as the static PoA. In general, this quantity gives an indication which processes are most conducive to spending large fractions of time in configurations that minimize overall cost (or maximize positive payoffs). We also note that in principle, the evolutionary PoA can be both smaller or larger than the static PoA.

Note that the concept of an evolutionary price of anarchy is neither bound to a particular game nor a particular evolutionary dynamics: it can serve as a general framework for analyzing a rich variety of dynamic games with arbitrary update dynamics.

4 Results and Analysis

In the following, we will consider pure strategies and the analysis of a setting with positive mutation rate $\mu > 0$. We will first show how to exactly calculate the selection-mutation equilibrium $\mathbf{x}$ of the evolutionary process for two fundamental graphs, and then use this to show how the ePoA can differ from a static analysis.

4.1 Analytical results

For simple graphs and pure strategies, $a_i \in \{0, 1\}$, we can calculate the stationary distribution of the underlying Markov chain under the different dynamics. We consider two instructive cases here (similar cases have been studied also in [2]), situated at the opposite ends of the spectrum: the *clique* (which results in perfectly global information) and the *star* (which is fully local). In these cases, we find exact results.

4.1.1 Clique

In a clique, the $(N + 1)$ states of the Markov chain are $i = 0, \ldots, N$, denoting the number of currently inoculated nodes. Here, the threshold for infection of an arbitrary insecure node is 1, as an infected node automatically spreads the virus to all other not inoculated nodes. We use the entries of the cost vector $ct = [I, V, L = 0]$ as the realized negative payoffs of infected (I), inoculated (V) and unaffected $(L = 0)$ nodes. For the expected payoffs $\hat{\pi}^i_X$ of nodes using strategy X, in the state i, with X either $C\,(a_i = 1)$ or $D\,(a_i = 0)$, we then find the following simple expressions:

$$\hat{\pi}^i_C = V, \text{ and } \hat{\pi}^i_D = \frac{i}{N}L + \frac{N-i}{N}I = \frac{N-i}{N}I \tag{6}$$

The next step is calculating the transition probabilities of the resulting Markov chain, $p_{i,i+1}$ and $p_{i,i-1}$, describing changes from states i to $i + 1$ or $i - 1$ (see [22] for details; for a more thorough introduction to the Moran process in evolutionary games, see e.g. [26]). To make the chain ergodic, we include mutation terms with $\mu > 0$ that prevent states 0 and n from being absorbing. Mutation means that with probability $\frac{1}{2}\mu$, the state switches. The expressions for these probabilities including mutation are as follows:

$$P_{i,i+1} = \frac{N-i}{N}\mu\frac{1}{2} + (1-\mu)p_{i,i+1}, \text{ and } P_{i,i-1} = \frac{i}{N}\mu\frac{1}{2} + (1-\mu)p_{i,i-1} \tag{7}$$

The terms $p_{i,j}$, i.e. the probabilities of state transitions without mutation, depend on which dynamics are used. That is, we will find different expressions for a Moran process as compared to a pairwise comparison. One caveat is that the virus inoculation game leads to expected payoffs $\hat{\pi}^i_X \leq 0$. To be able to use these terms in the equations for the Moran process probabilities, we use the standard assumption of an *exponential fitness function* (see [28]): expected payoffs are mapped to a fitness with the function $F(x) = e^x$, such that the fitness becomes $f^i_X = e^{s\hat{\pi}^i_X}$. We subsequently set the parameter $s = 1$, as is common in the literature. This quantity is now always positive, is larger for smaller costs (or equivalently, larger payoffs), and can be used in the standard Moran probabilities (cf. [18] and [22]).

Meanwhile, for the pairwise comparison – imitation dynamics, we can still use the payoffs themselves without transforming them (see [22]).

From these transition probabilities, we can calculate the stationary distribution of the process: it is the normalized left eigenvector of the transition matrix, which is the tridiagonal matrix $\mathbf{P}$, see [22].

The mutation-selection equilibrium is then the solution to

$$\mathbf{xP} = \mathbf{x}.$$

It is the limit distribution (which exists as per the ergodicity of the underlying Markov chain) that gives the long-run frequencies of all possible states. These frequencies can also be obtained by simulating the process for long enough and counting how often each state occurs.

To be able to compare the evolutionary price of anarchy with the static price of anarchy, we first need to describe the Nash equilibria of the system. For the complete graph, by using Corollary 1.1, the static analysis predicts one equivalence class of Nash equilibria, $\mathcal{N}$, where exactly $N - t = N - VN/I$ nodes are inoculated. We denote the efficiency of these equilibria as PoA, the static price of anarchy.

In order to calculate $ePoA$, we first calculate the average social cost $\hat{S}$. We do so either by averaging over the total cost in each round in our simulations, or taking $\hat{S} = \mathbf{xR}$ (cf. Eq. 4). For the complete graph, the vector $\mathbf{R}$ containing the total system cost in all possible configurations, with $i = 0, ..., N$ inoculated nodes, has the components

$$R_i = i\hat{\pi}^i_C + (N - i)\hat{\pi}^i_D = iV + (N - i)\frac{N - i}{N}I. \tag{8}$$

We also know the cost of the optimum; it is attained in the state with $i^* = \frac{N(2L-V)}{2L}$, which is the number of inoculated nodes where $R^*_i = \max R_i$ holds. The optimal cost is then $\Omega = i^*V + \frac{(N-i^*)^2}{N}I$. With this, we can use Eq. 5 to measure the efficiency of the different dynamics by finding their corresponding evolutionary price of anarchy as $\hat{S}/\Omega$. We present our analysis below.

Using our evolutionary process, our findings can now be summarized in the following lemma:

▶ **Lemma 2.** *For a fixed cost vector $ct = [V, I, 0]$, large $N \gtrapprox 30$, any reasonable mutation rates $0 < \mu < 0.5$, and intermediate to large selection strength $\beta > 1$ (for the pairwise comparison process), we always recover the predicted equivalence class of Nash equilibria, $\mathcal{N}$, as the most abundant state in the selection-mutation equilibrium of both types of processes. That is, the process spends the largest fraction of time in the Nash equilibria, where exactly $t = VN/I$ nodes are inoculated.*

We note that there is also substantial weight on neighboring states of the Nash equilibria (with $t \pm i$, where $i \in \{1, 2, 3, ...\}$) with worse total welfare, due to the stochasticity of the process. However, the average social cost $\hat{S}$ is not substantially different from the cost of the Nash equilibria, since the weight on neighboring states is symmetrically distributed. The numerical results these insights are based on are provided in Figs. 3 and 4, as well as in [22] and upon request in full.

What this means for the evolutionary price of anarchy is expressed in the following corollary.

▶ **Corollary 3** (Evolutionary Price of Anarchy for Cliques.). *The evolutionary price of anarchy $ePoA_{Clique}$ in a clique with N nodes approaches the efficiency of the Nash equilibrium (the price of anarchy PoA_{Clique}), as $\mu \to 0$ and N grows large, such that $\lim_{N\to\infty} |ePoA_{Clique} - PoA_{Clique}| = 0$.*

A straightforward argument shows why Lemma 2 and Corollary 3 hold.

For the base case of the unstructured population on a complete graph, the perfect information setting corresponds to the local information setting, as each node only has distance 1 from every other node. Furthermore, the Markov chain underlying the stochastic evolutionary process is ergodic by $\mu > 0$, such that there exists a stationary distribution and the process converges to it. This stationary distribution then places most weight on the Nash equilibrium described in [2], as it is the configuration where nodes have no incentive to switch their strategy, and are aware of this fact, just as in the static case. The stochastic noise becomes smaller as the number of nodes grows larger (which inhibits the system spending too much time in the two extremal states of all nodes inoculated and all nodes insecure), and as the mutation rate becomes smaller (which sharpens the peak of the distribution at the equilibrium). This lets us recover the equilibrium results of [2].

4.1.2 Star graph

For star graphs $K_{1,N-1}$, we can also numerically compute the expected payoffs and the Markov chain properties of the process. The $2N$ states in this case are of the form (t, l), $t \in \{0, 1\}$ and $l \in \{0, \dots, N-1\}$. The parameter t denotes the inoculation state of the center node, whereas l gives the number of inoculated leaf nodes.

By using Corollary 1.1 again, we find two equivalence classes of Nash equilibria: class $\mathcal{N}_1$ has $N - t = N - \lfloor VN/I \rfloor$ inoculated leaf nodes (which in our notation is the state $(0, N-t)$), whereas $\mathcal{N}_2$ contains the optimal equilibrium, which features the center as the only inoculated node (state $(1, 0)$). We will show that for this highly structured population, the outcome can be quite different from the predictions of the static analysis of the one-shot game. Local evolutionary processes prevent the system from spending too much time in either of the equilibria classes. We detail this in the following paragraphs.

To see this, we first compute the expected payoffs of leaf nodes ($\hat{\pi}_X^{t,l}$, with $X \in \{C, D\}$) and the center node ($\hat{\pi}_{Center}^{t,l}$) in the configurations (t, l):

$$\hat{\pi}_C^{t,l} = \hat{\pi}_{Center}^{1,l} = V, \ \hat{\pi}_D^{0,l} = \hat{\pi}_{Center}^{0,l} = \frac{N-l}{N} I, \text{ and } \hat{\pi}_D^{1,l} = \frac{1}{N} I. \tag{9}$$

For the Moran and pairwise comparison – imitation dynamics we derive probabilities $p_{n,o}^{k,m}$ that describe the transitions $(k, n) \to (m, o)$ without mutation, again with fitness $f_X^{t,l} = e^{\hat{\pi}_X^{t,l}}$. The exact expressions can be found in [22].

We can now get the transition matrix, with its entries $P_{l,l}^{0,1}$ [22], and subsequently the selection-mutation equilibrium $\mathbf{x}$ of the process with its corresponding average system cost.

Having calculated/simulated the stationary distribution, we observe the following (see [22]):

- No matter the network size or the process, the Nash equilibria in $\mathcal{N}_1, \mathcal{N}_2$ are not abundant states in the stationary distribution, with up to a factor 10^3 in weight difference to the more frequent states. We instead find a high abundance of costly non-equilibrium states $\mathcal{X} = \{(0, N-t-i)\}$ for some integers $t > i > 0$. There is also substantial weight on the beneficial configurations with low cost $(1, N-t-i)$ for the same values of i.
- The equilibrium $\mathcal{N}_1 = (0, N-t)$ is typically of far lower frequency than the non-equilibrium states . But it still plays more of a role overall than the optimum $\mathcal{O} = \mathcal{N}_2 = (1, 0)$, which is a rare state at stationarity and almost never visited in the long run.

We will now argue why the process exhibits this off-equilibrium behavior. First of all, starting from the above observations, it is straightforward to show why the optimum, that is, the state $\mathcal{O} = (1, 0)$, cannot be an abundant state in the mutation-selection equilibrium of a star graph.

▶ **Lemma 4** (The optimal Nash equilibrium is rare in the mutation-selection equilibrium.). *Consider a star graph, with fixed but arbitrary number of nodes N. For arbitrary mutation rates $\mu > 0$, arbitrary $|V| < |I|$, and any of the three evolutionary processes we consider, the optimal Nash equilibrium $\mathcal{O} = (1, 0)$ cannot be an abundant state in the mutation-selection equilibrium.*

Proof. In fact, the equilibrium is not even reached if not for mutation. To see this, consider the states $a = (0, 0)$ and $b = (1, 1)$, and suppose $\mu = 0$. While both these states only differ in one node's strategy from the Nash equilibrium, they cannot serve as starting points for a transition. State a is absorbing in this scenario, as there is no inoculated node to pass on its strategy. Meanwhile, in the state b, the one inoculated leaf node cannot change its strategy without the center node being not inoculated – the terms $p^{1,1}_{l,l-1}$ are always zero. This however leads to the opposite of the Nash equilibrium we are trying to reach. Thus, only a nonzero mutation rate can provide access to this state. At the same time, it is clear that the transitions from $\mathcal{O}$ do not need mutation to reach the neighboring states a and b, which leads to a higher probability to leave the state than to reach it. This makes $\mathcal{O}$ unsustainable in the long run. ◀

The following lemma states the corresponding result for the other Nash equilibria.

▶ **Lemma 5** (The Nash equilibria from the equivalence class $\mathcal{N}_1$ cannot form an abundant state in the mutation-selection equilibrium.). *Consider a star graph, with fixed but arbitrary number of nodes N. For arbitrary mutation rates μ, arbitrary $|V| < |I|$, and any of the three evolutionary processes we consider, the Nash equilibria of the form $(0, N-t)$ cannot be an abundant state in the mutation-selection equilibrium. Instead, pairs of states of the form $(0, N-t-i)$ and $(1, N-t-i)$, $i \in \mathcal{I}$ with the set $\mathcal{I} \subseteq \mathbb{N}_0$ depending on N, V, I, μ and the evolutionary dynamics used, act as sinks of the process.*

We illustrate our argument by Fig. 2, where we show the case of the pairwise comparison process on a star graph with $N = 12$ and $V/I = 1/3$.

Consider the process as a two-dimensional random walk, defined by our transition probabilities $p^{k,m}_{n,o}$ (see [22]) in the limit of the mutation rate $\mu \to 0$. Let $t = VN/I$; the Nash equilibrium $\mathcal{N}_1$ is then the state $(0, N-t)$, as discussed above. In the example, we have $t = 4$, such that $\mathcal{N}_1 = (0, 8)$.
For easier readability, we use the notation $\{u_{(i,i+1)}, q_{(i,i-1)}, r_i, s_i\}$ for $\{p^{0,0}_{N-t-i,N-t-i-1}, p^{1,1}_{N-t-i,N-t-i+1}, p^{0,1}_{N-t-i,N-t-i}, p^{1,0}_{N-t-i,N-t-i}\}$ (cf. [22]). That is, u gives the probability of moving one further step away from the Nash equilibrium $\mathcal{N}_1$ by one leaf node

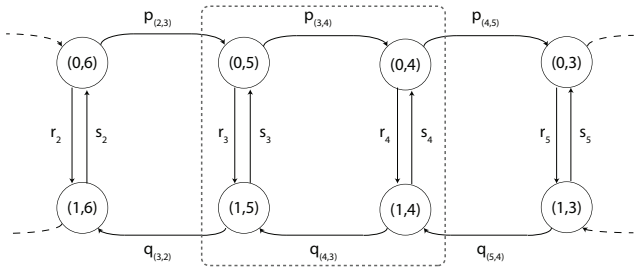


Figure 2 Crucial section of the Markov chain underlying evolutionary processes with local information on the star graph $K_{1,11}$. The ratios r/u, s/q, and u/q of transition probabilities and their dependency on the distance i from the Nash equilibrium $(0, 8)$ has the two-dim. random walk oscillate between pairs of states $\{(0, 5), (1, 5)\}$ and $\{(0, 4), (1, 4)\}$, and thus be trapped in increasingly constricted cycles.

switching to insecure; q the probability of moving one step closer to $\mathcal{N}_1$ by one leaf node switching to secure; r the probability of the center switching to secure, and s the probability of the center switching to insecure. The parameter i can be thought to be the distance to $\mathcal{N}_1$. Note that this two-dimensional random walk has a defined direction; there is no possibility to increase i in the lower level of the chain (where the center is inoculated), and no possibility to decrease i in the upper level (where the center is insecure).
We show in [22] that due to these transition probabilities, the random walk gets trapped in increasingly constricting cycles as it moves away from $\mathcal{N}_1$. In the example of Fig. 2, these states form the set $\{(0, 5), (0, 4), (1, 5), (1, 4)\}$, with the most weight on $(0, 4)$ and $(0, 5)$.

We thus have shown that using local information only, the system spends a high fraction of time in states that are not Nash equilibria, and will not reach the optimum. What does this mean for the evolutionary price of anarchy?

Seeing that the abundant states carry a high social cost compared to the optimum and also the worse equilibria in $\mathcal{N}_1$, it is already intuitive that $ePoA$ will be larger than PoA in the star graph, as long as the mutation rate μ is sufficiently small ($\mu \lessapprox 0.005$). Indeed, using the stationary distribution $\mathbf{x}$ to calculate the social cost $\hat{S}$ as in Eq. 4, and then the evolutionary price $ePoA$ as in Eq. 5, we find that the evolutionary process has to settle for a relatively high $\hat{S}$, and with it, a high $ePoA$ (see also Figs. 3- 4). We summarize this in the following corollary:

▶ **Corollary 6** (Evolutionary Price of Anarchy for Star Graphs.). *For small mutation rates $\mu \lessapprox 0.005$, arbitrary N and arbitrary $|V| < |I|$, the evolutionary price of anarchy $ePoA_{Star}$ in a star graph $K_{1,N-1}$ with N nodes is at least equal to or higher than the static price of anarchy PoA_{Star}. That is, $ePoA_{Star} - PoA_{Star} \geq 0$.*

We note that the exact evolutionary price in relation to the static price of anarchy is determined by parameter choices. This means that with a mutation rate $\mu \gtrapprox 0.005$, for some choices of cost vector and network size, it is possible to achieve a slightly lower *average* cost than can be achieved by the worst possible solution to the one-shot game due to higher mutation facilitating contributions from states that lead to a high total payoff (see Fig. 3).

However, our results let us conjecture that for a local information model with reasonably small mutation rates, we cannot hope to do much better on average than the worst Nash equilibrium in highly structured networks (like a star), much less reach the optimum, such that paying (at least) the price of anarchy is not only a theoretical possibility, but also a realized fact.

5 Simulation of more complex topologies

With the algorithms introduced above, we are able to simulate the process also for more complicated graphs. While numerical analysis is usually impossible in these cases - it is intuitive that obtaining numerically precise results on the stochastic process will be harder with increasing graph size and less inherent symmetry in the graph we can always compute the average social welfare by simulating the different dynamics long enough. We can even feasibly find the stationary distribution for graphs that have some inherent symmetry.

5.1 2-clique network

This graph consists of two cliques, connected by a single path. The states of the Markov chain are now (a, b, c, d), with $a, c \in \{0, 1\}$ denoting the inoculation state of the two nodes where the cliques are joined (subsequently called the hubs), and $b, d \in \{0, ...N-1\}$ denoting how many of the $N-1$ remaining nodes on each side are inoculated. For this graph, seeing that the underlying Markov chain of the process only has N^2 states, it is indeed possible to also find both the selection-mutation equilibrium and the average social cost $\hat{S}$ by simulation. To be able to calculate $ePoA$, and compare it to the efficiencies of other equilibria as well as the static price of anarchy PoA, we also compute the social welfare in all possible configurations (vector $\vec{R}$), and find the optimum as the maximum over $\vec{R}$. This section describes our findings after running our simulations with $\mu = 0.001$ for the network sizes $N = \{10, 12, 20, 50\}$ and the realized cost vectors $ct = \{[-2, -1, 0], [-3, -1, 0]\}$.

1. First considering static Nash equilibria as the baseline, we again use Corollary 1.1 with $t = V/I$ and find two equivalence classes $\mathcal{N}_1, \mathcal{N}_2$ of Nash equilibria. Class $\mathcal{N}_1$ contains the states $(0, p, 0, q)$, with $p + q = N - t$, which have both hubs insecure, and p respective q inoculated nodes in the remainders of the two cliques. The other class of equilibria, $\mathcal{N}_2$, is composed of the states $(1, N/2 - t - 1, r, s)$ and $(r, s, 1, N/2 - t - 1)$ with $r + s = N/2 - t$, which are the states where at least one hub is always inoculated. In the case where N is divisible by I, the two equilibria classes are equivalent, giving the same cost. However, when N is not divisible by I, $\mathcal{N}_2$ is more efficient with respect to the overall cost, such that it is the cost of the equilibria in $\mathcal{N}_1$ which is used to calculate the PoA.
2. In evolution, we find an $ePoA > 1$ for all three network sizes and all three evolutionary processes – the optimum is not an abundant state. It is also the case that $ePoA \geq PoA$, making the average cost slightly higher than the static price of anarchy, even though $ePoA$ becomes smaller with increasing network size. In all the tested scenarios, the system does not spend a substantial fraction of time close to social optima and does not make up for partially costly off-equilibrium behavior it exhibits otherwise. We again find that the Moran processes show behavior that is slightly dependent on the network size N, whereas the pairwise comparison process gives more consistent results even for smaller N, and also exhibits the smallest value of $ePoA$.

We have seen in Corollary 3 that for one clique on its own, we recover the Nash equilibria in our evolutionary process. However, the behavior of such a process on the 2-clique graph leads to an outcome that differs both from the optimum and the static predictions for Nash equilibria. Intuitively, this is due to the link between the two hubs acting as a bottleneck for information flow.

5.2 2-star network

Another example of a symmetric graph is two stars, joined at their hubs. Here, we find:

1. We again begin with the static Nash equilibria and the static PoA. In this case, there are three equivalence classes of equilibria: $\mathcal{N}_1$ corresponds to its counterpart in the 2-clique graph – it contains states $(0, p, 0, q)$ (defined analogously to above), with $p + q = N - t$, and is again the class with the most costly equilibria. It therefore is used for the static price of anarchy. Class $\mathcal{N}_2$ consists of the states $(1, 0, s, 0)$, with $s = 0$ if $t \geq N/2$ and $s = 1$ otherwise. Lastly, $\mathcal{N}_3$ exists if $t < N/2$, and features states of the type $(1, 0, 0, N/2 - t + 1)$. Here, we can also explicitly characterize the optimum: it is always the state $O = (1, 0, 1, 0)$.
2. An evolutionary process on the 2-star graph for the network sizes and V/I ratios tested again leads to an $ePoA \geq PoA$. Seeing that the basic component of the graph – the simple star – already exhibits off-equilibrium behavior, this is not too surprising. However, as opposed to the single star, we now observe that the Moran Birth-Death scenario is advantageous for all network sizes, as it leads to the lowest overall $ePoA$.

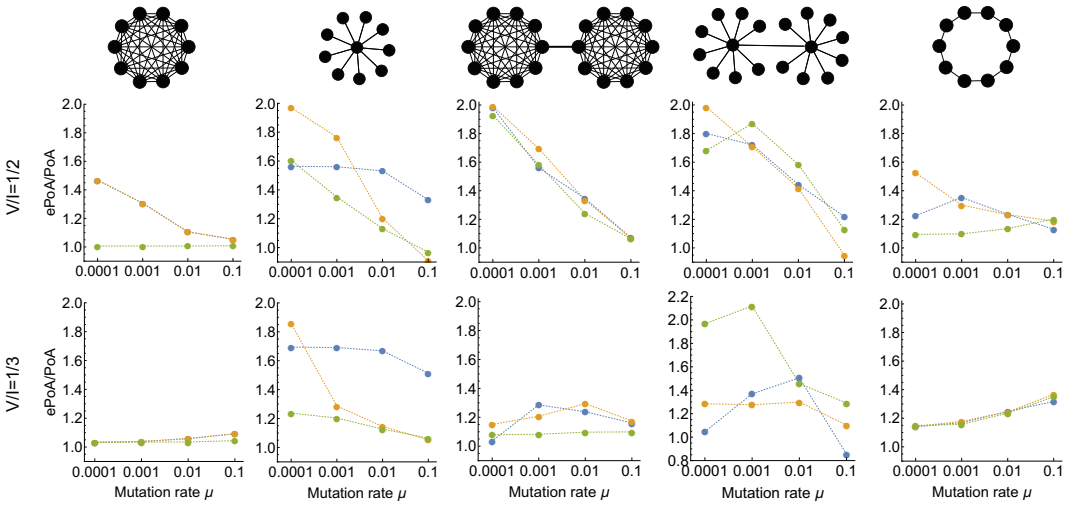


Figure 3 We visualize the ratio of the evolutionary price of anarchy to the price of anarchy, $ePoA/PoA$, for the four discussed topologies, two different values of V/I, and varying *mutation rate*. The network size is kept constant at $N = 20$. We plot the three different evolutionary dynamics together: Moran-DB (blue), Moran-BD (yellow), and pairwise comparison process (green). We see that different processes show different efficiency, depending on topology and mutation rate, and the behavior of the $ePoA$ does not have to be strictly monotonic - there can be "sweet spots" for certain combinations of parameters. Simulations were run for $5 * 10^5$ iterations.

5.3 Cycle graph

Another structure of interest is the cycle. While this topology is quite simple, the process does not show straightforward behavior due to a large state space with few possible transitions.

Due to symmetry considerations, we can encode states corresponding to configurations $\mathbf{a} \in \{0, 1\}^N$ by counting the number of insecure nodes in between every two inoculated nodes, and listing the number of gaps ordered by their size, starting at 1. As an example, we can take a network of size $N = 6$ with $i = 4$ inoculated nodes in the configuration 010111,

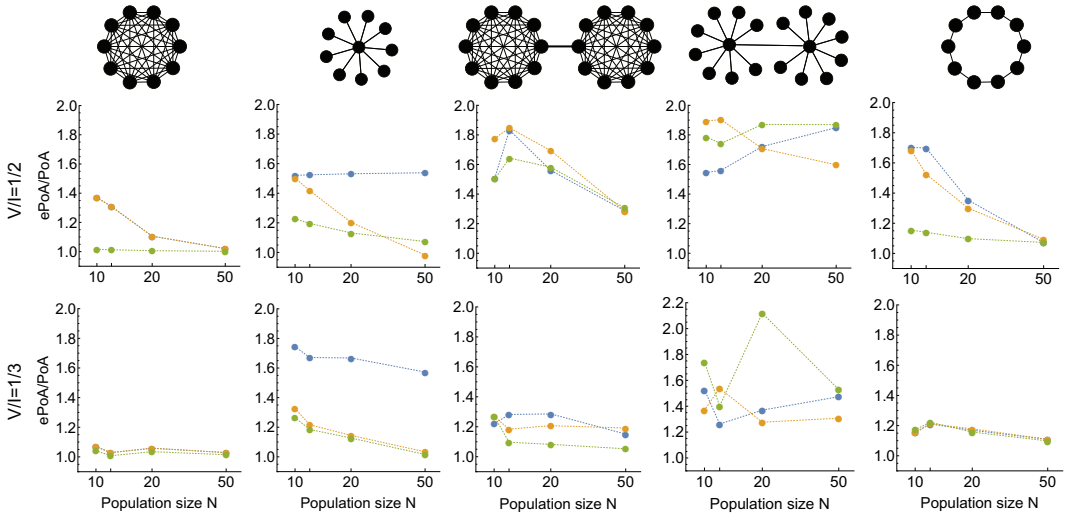


Figure 4 We visualize the ratio of the evolutionary price of anarchy to the price of anarchy, $ePoA/PoA$, for the four discussed topologies, two different values of V/I, and varying *network size*. The mutation rate is kept constant at $\mu = 0.001$. We again plot the three different evolutionary dynamics together: Moran-DB (blue), Moran-BD (yellow), and pairwise comparison process (green). We see that different processes show different efficiency, depending on the network topology and the network size. Again, the behavior of the $ePoA$ does not have to be strictly monotonic - there can be "sweet spots" for certain combinations of parameters. Simulations were run for $5 * 10^5$ iterations.

which corresponds to the state $[4, 2, 0, 0, 0, 0, 0]$, where the first number in the list is simply $i = 4$. This is thereby also equivalent to the configurations 101011, 110101 and 111010. To enumerate and order these states, we can describe the sets of states with i inoculated nodes more generally as $\{i\}$-integer partitions of $N - i$, where an $\{i\}$-partition of x is the set of partitions of x with length $\leq i$. This way, we get a natural ordering on states - from the smallest partition length to the largest-, and find a simple way to enumerate them. In the previous example, the corresponding partition of $N - i = 2$ is $(1, 1)$. The total number of states in a network with N nodes is then $2 + \sum_{i=1}^{N-1} \sum_{k=1}^{i} p_k(N - i)$, where $p_k(x)$ is the partition function, i.e. the number of integer partitions of x with length k.

For a state to be a Nash equilibrium, the largest insecure component cannot be larger than $t = VN/I$. For this condition to hold, the number of inoculated nodes has to be $i^* = \left\lceil \frac{N}{t+1} \right\rceil$, and the Nash equilibrium states is then the $\{i\}$-partition of $N - i^*$, where the largest part is at most t. In our toy example above, using $V/I = 1/2$, those are the states $[2, 1, 0, 1, 0, 0, 0]$ and $[2, 0, 2, 0, 0, 0, 0]$.

We run simulations for $N = \{10, 12, 20, 50\}$ and $\mu = \{0.0001, 0.001, 0.01, 0.1\}$, and again find the $ePoA$ to be higher than the PoA, making the cycle a suboptimal topology for efficient inoculation: clusters of inoculated/insecure nodes can not be broken if not for mutation. Intuitively speaking, a Nash equilibrium with e.g. 2 inoculated nodes that are not next to each other will not easily be reached, as moving an inoculated node from point A to B has to involve mutation: starting from a cluster of two inoculated nodes, two imitation events and two mutation events are necessary to move the second node two steps away from the first one. This becomes more unlikely as $\mu \to 0$. However, a surprising result can be seen in Fig. 3, for $V/I = 1/3$: the three evolutionary processes behave more or less the same in this case, and the efficiency is monotonic in the mutation rate, just as in the complete graph. An interesting direction for future work would thus be a more thorough analysis of the role of regular graphs in the behavior of stochastic evolutionary processes.

6 Related Work

There exists much research on alternative equilibrium concepts which overcome the shortcomings of Nash equilibria. An interesting concept in this context are Bayes-Nash equilibria [23] in games of incomplete information (e.g., [23]), and in particular the Bayes-Nash Price of Anarchy [20]. Another interesting concept, also based on dynamic equilibria, is the stochastic price of anarchy [6]: it addresses the issue that some PoAs are sensitive to small perturbations, by considering more stable concepts. However, these models come with strong assumptions on the agents' information and/or resources. In particular, not much is known today about the price of anarchy in games with partial information *based on local interactions*. The stochastic price of anarchy even requires agents to have information beyond the current game (which is derived implicitly), to decide on their best action: the players need knowledge of the execution *history*. Dynamic games based on learning dynamics [4], such as regret minimization [11] or fictitious play [12], have the same shortcoming: they require players to keep track of historical information. There is work on network creation games with local knowledge [3], however, it only considers the static pice of anarchy and does not embed the game into an evolutionary process. All these results hence do not apply to the evolutionary games studied in this paper. Indeed, to the best of our knowledge, this paper is the first to consider the application of the price of anarchy to evolutionary games where players are memoryless and only have (graph-)local information.

7 Conclusion

This paper introduced the evolutionary price of anarchy to study equilibrium behavior of simple agents interacting in a distributed system, based on local information. We showed that for memoryless agents, the resulting equilibria can be significantly different from their static equivalent, and that Nash equilibria are sometimes assumed only very infrequently.

We believe that our work can provide a novel perspective on the discussion of the impact of limited information in games. In particular, it opens several interesting avenues for future research. Our model still comes with several limitations due to the well-known notorious difficulty of analyzing evolutionary multi-player games on graphs. In particular, it will be interesting to analyze the ePoA for additional topologies and more general interaction models (beyond memoryless), as well as to explore randomized (mixed) strategies. It would also be interesting to prove or disprove our conjecture that processes based on imitation dynamics always result in a ePoA which is higher than the PoA in the virus inoculation game.

References

1. Benjamin Allen and Martin A Nowak. Games on graphs. *EMS surveys in mathematical sciences*, 1(1):113–151, 2014.
2. James Aspnes, Kevin Chang, and Aleksandr Yampolskiy. Inoculation Strategies for Victims of Viruses and the Sum-of-squares Partition Problem. In *Proc. 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 43–52, Philadelphia, PA, USA, 2005.
3. Davide Bilò, Luciano Gualà, Stefano Leucci, and Guido Proietti. Locality-based Network Creation Games. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, pages 277–286, New York, NY, USA, 2014. ACM.
4. Nicolo Cesa-Bianchi and Gabor Lugosi. *Prediction, learning, and games*. Cambridge university press, 2006.
5. Krishnendu Chatterjee and Thomas A Henzinger. A survey of stochastic ω-regular games. *Journal of Computer and System Sciences*, 78(2):394–413, 2012.

6 Christine Chung, Katrina Ligett, Kirk Pruhs, and Aaron Roth. The price of stochastic anarchy. In *International Symposium on Algorithmic Game Theory*, pages 303–314. Springer, 2008.

7 Alex Fabrikant, Ankur Luthra, Elitza Maneva, Christos H Papadimitriou, and Scott Shenker. On a network creation game. In *Proc. 22nd annual symposium on Principles of distributed computing (PODC)*, pages 347–351. ACM, 2003.

8 Chris Fleizach, Michael Liljenstam, Per Johansson, Geoffrey M Voelker, and Andras Mehes. Can you infect me now?: malware propagation in mobile phone networks. In *Proceedings of the 2007 ACM workshop on Recurring malcode*, pages 61–68. ACM, 2007.

9 Drew Fudenberg and Lorens A. Imhof. Monotone imitation dynamics in large populations. *Journal of Economic Theory*, 140(1):229–245, 2008.

10 Drew Fudenberg, Martin A Nowak, Christine Taylor, and Lorens A Imhof. Evolutionary game dynamics in finite populations with strong selection and weak mutation. *Theoretical population biology*, 70(3):352–363, 2006.

11 Sergiu Hart and Andreu Mas-Colell. A simple adaptive procedure leading to correlated equilibrium. *Econometrica*, 68(5):1127–1150, 2000.

12 Josef Hofbauer and William H Sandholm. On the global convergence of stochastic fictitious play. *Econometrica*, 70(6):2265–2294, 2002.

13 Josef Hofbauer and Karl Sigmund. *Evolutionary games and population dynamics*. Cambridge university press, 1998.

14 Elias Koutsoupias and Christos Papadimitriou. Worst-case equilibria. In *Annual Symposium on Theoretical Aspects of Computer Science (STOC)*, pages 404–413. Springer, 1999.

15 Erez Lieberman, Christoph Hauert, and Martin A Nowak. Evolutionary dynamics on graphs. *Nature*, 433(7023):312, 2005.

16 Thomas Locher, Patrick Moor, Stefan Schmid, and Roger Wattenhofer. Free Riding in BitTorrent is Cheap. In *Proc. 5th Workshop on Hot Topics in Networks (HotNets)*, pages 85–90, 2006.

17 Swarup Mohalik and Igor Walukiewicz. Distributed Games. In *FST TCS 2003: Foundations of Software Technology and Theoretical Computer Science*, pages 338–351. Springer, 2003.

18 Martin A. Nowak. *Evolutionary Dynamics*, pages 123–143. Harvard University Press, 2006.

19 Amir Pneuli and Roni Rosner. Distributed reactive systems are hard to synthesize. In *Proc. 31st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 746–757, 1990.

20 Tim Roughgarden. The price of anarchy in games of incomplete information. *ACM Transactions on Economics and Computation (TEAC)*, 3(1):6, 2015.

21 Tim Roughgarden and Éva Tardos. How bad is selfish routing? *Journal of the ACM (JACM)*, 49(2):236–259, 2002.

22 Laura Schmid, Krishnendu Chatterjee, and Stefan Schmid. The Evolutionary Price of Anarchy: Locally Bounded Agents in a Dynamic Virus Game. *arXiv preprint arXiv:1906.00110*, 2019.

23 Satinder Singh, Vishal Soni, and Michael Wellman. Computing approximate Bayes-Nash equilibria in tree-games of incomplete information. In *Proc. 5th ACM conference on Electronic Commerce (EC)*, pages 81–90. ACM, 2004.

24 J Maynard Smith and George R Price. The logic of animal conflict. *Nature*, 246(5427):15, 1973.

25 John Maynard Smith. Evolution and the Theory of Games. In *Did Darwin Get It Right?*, pages 202–215. Springer, 1988.

26 Christine Taylor, Drew Fudenberg, Akira Sasaki, and Martin A Nowak. Evolutionary game dynamics in finite populations. *Bulletin of mathematical biology*, 66(6):1621–1644, 2004.

27 Arne Traulsen, Martin A. Nowak, and Jorge M. Pacheco. Stochastic dynamics of invasion and fixation. *Phys. Rev. E*, 74:011909, July 2006.

28 Arne Traulsen, Noam Shoresh, and Martin A Nowak. Analytical results for individual and group selection of any intensity. *Bulletin of mathematical biology*, 70(5):1410, 2008.

29 Jörgen W Weibull. *Evolutionary game theory*. MIT press, 1997.

Tight Bounds on Distributed Exploration of Temporal Graphs

Tsuyoshi Gotoh
Osaka University, Japan

Paola Flocchini
University of Ottawa, Canada

Toshimitsu Masuzawa
Osaka University, Japan

Nicola Santoro
Carleton University, Canada

Abstract

Temporal graphs (or evolving graphs) are time-varying graphs where time is assumed to be discrete. In this paper, we consider for the first time the problem of exploring temporal graphs of *arbitrary unknown topology*. We study the feasibility of exploration, under both the Fsync and Ssync schedulers, focusing on the number of agents necessary and sufficient to explore such graphs.

We first consider the minimal (i.e., less restrictive) assumption on the dynamics of the graph under which exploration is still feasible: *temporal connectivity*. Let $\mathcal{H}$ be the class of temporally connected graphs; we show that for any temporal graph $\mathcal{G} \in \mathcal{H}$ the number of agents sufficient to perform exploration is related to the number of its transient edges, a parameter $\eta(\mathcal{G})$ we call evanescence of the graph. More precisely, any $\mathcal{G} \in \mathcal{H}$ can be explored by a team of $k \geq 2\eta(\mathcal{G}) + 1$ agents; this bound is tight as we prove there are $\mathcal{G} \in \mathcal{H}$ that cannot be explored by $2\eta(\mathcal{G})$ agents.

We then turn our attention to the well-known stronger assumption on the dynamics of the graph, called *1-interval connectivity*: the graph is connected at any time step. Let $\mathcal{W} \subset \mathcal{H}$ be the class of these always-connected temporal graphs. For this class, we prove the existence of a difference between Fsync and Ssync when there is a bound ℓ on the number of edges missing at each time. In fact, we show a tight bound of $2\ell + 1$ on the number of agents necessary and sufficient in Ssync, and a smaller tight bound of 2ℓ in Fsync. As a corollary, we re-establish two recently published bounds for 1-interval connected rings.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Theory of computation → Distributed algorithms

Keywords and phrases Distributed algorithm, Mobile agents, Exploration of dynamic networks, Arbitrary footprint

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.22

Acknowledgements This research was partly supported by NSERC through the Discovery Grant program, by Prof. Flocchini's University Research Chair, and JSPS KAKENHI Grant Number 19H04085.

1 Introduction

1.1 Framework and Background

The *graph exploration* problem (Exploration), first introduced by Shannon [34], is a fundamental problem in theoretical computer science, in particular in the field of distributed computing by mobile entities. It requires each node of the graph to be visited by one or more entities, called agents, a finite number of times (exploration *with termination*) or infinitely often (*perpetual* exploration). In addition to its theoretical importance, Exploration is

© Tsuyoshi Gotoh, Paola Flocchini, Toshimitsu Masuzawa, and Nicola Santoro;
licensed under Creative Commons License CC-BY
23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller; Article No. 22; pp. 22:1–22:16
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

relevant from a practical viewpoint in networks with mobile entities (e.g., software agents, vehicles, or robots): by visiting all nodes, agents can check whether there are some nodes with problems in the network, propagate some data across the network, or collect (or search) specific information from the whole network.

This problem has been extensively studied over a variety of assumptions and settings depending on whether the nodes have distinct labelings or are anonymous, on the type of communication mechanisms available to the agents, on the degree of synchronization of the network, on the level of knowledge the agents have about the graph, on their memory, etc. (e.g., see [1, 8, 7, 10, 13, 14, 21, 22, 33, 35], and [9] for a recent survey). In spite of all the differences, the existing literature has until very recently made a common assumption: the graph is *static*, i.e., the link structure does not change during the exploration.

Recently, researchers in the distributed computing community have started to investigate *highly dynamic graphs* that are graphs where the topological changes are not sporadic or anomalous, but rather inherent in the nature of the network. Various models have been proposed to describe highly dynamic networks, under a variety of names. A model that describes them in a simple and natural way is the one of *time-varying graphs*, formally defined in [6], where main classes of systems studied in the literature and their computational relationship were identified. When time is assumed to be *discrete*, the evolution of these systems can be equivalently described as a sequence of static graphs, called *evolving graph* or *temporal graph*, a model suggested in [25], formalized in [17].

If the dynamics of the changes is arbitrary and unrestricted, clearly any non-trivial computation is unfeasible and any non-trivial problem is unsolvable. Hence, all the studies are carried out under some assumptions restricting the arbitrariness of the dynamics. The minimal (i.e., less restrictive) assumption is *temporal connectivity*: starting at any time, there is temporal reachability between any two nodes (e.g., [5]). Stronger assumptions include *1-interval connectivity* : the graph is always connected (e.g., [24, 30, 31]); and *T-interval connectivity* : the graph is always connected and every $T > 1$ consecutive rounds contain the same spanning-tree (e.g., [28, 30]). A classification of the most common assumptions was done in [6].

While there are several studies on computations by mobile agents moving in temporal graphs (for a recent survey see [11]), the results on the *exploration of temporal graphs* are rather limited. On the probabilistic side, there is an early seminal work on random walks [2]. On the deterministic side there are: the study of the complexity of computing a foremost exploration schedule under the 1-interval-connectivity assumption [32], generalized and extended in [15] and then in [16]; the computation of an exploration schedule for *rings* under the stronger T-interval-connectivity assumption [28]; the computation of an exploration schedule for *cactuses* under the 1-interval-connectivity assumption [26]. These studies are however *centralized* (or off-line); that is, they assume that the exploring agents have complete a priori knowledge of the topological changes and the times of their occurrence. *Distributed* approaches have been studied under particular constraints on the network connectivity and on its underlying topology. Exploration with termination by a single agent of periodic temporal networks, including *carrier networks*, has been studied in [18, 19, 27, 28]. Exploration with termination of 1-interval connected *rings* by two and three agents under both synchronous and semi-synchronous schedulers has been considered in [12]. Perpetual exploration by three agents on temporally connected *rings* has been studied in [4, 5]. Exploration with termination by $O(n)$ agents of $n \times m$ dynamic *tori* ($n \leq m$), where each column and row is a 1-interval connected ring, has been investigated in [23].

All the existing results on distributed exploration of time-varying graphs have been obtained for temporal graphs with very specific topologies (rings, tori, or collections of cycles in the case of carrier networks). In this paper we start the investigation of the exploration of temporal graphs with *arbitrary* and *unknown* topologies.

1.2 Contributions

In this paper we consider perpetual exploration of time varying graphs whose topology is arbitrary and unknown to the agents. We focus on solvability of the exploration of such dynamic graphs and we determine the number of agents that are necessary and sufficient for exploration under the FSYNC and SSYNC activation schedulers.

Clearly, if the graph is not *temporally connected*, perpetual exploration is trivially impossible to achieve. We thus start our investigation with the class $\mathcal{H}$ of temporally connected temporal graphs. We show that for the graphs $\mathcal{G} \in \mathcal{H}$, the number of agents sufficient to perform exploration is related to the evanescence $\eta(\mathcal{G})$ of the graph, that is the number of transient edges. More precisely, any $\mathcal{G} \in \mathcal{H}$ can be explored by a team of $k \geq 2\eta(\mathcal{G}) + 1$ agents; this bound is tight as we prove there are $\mathcal{G} \in \mathcal{H}$ that cannot be explored by $2\eta(\mathcal{G})$ agents. The impossibility holds under very strong conditions (FSYNC scheduler, agents and nodes with distinct IDs, knowledge on n and k). On the other hand, the proposed exploration algorithm, based on the rotor router technique, works under very weak conditions (SSYNC scheduler, anonymous agents, no knowledge of topological parameters).

We then turn our attention to the stronger assumption on the dynamics of the graph, *1-interval connectivity*: the graph is always connected. Let $\mathcal{W}(\ell) \subset \mathcal{H}$ be the class of these always-connected temporal graphs where the number of missing edges at each time is at most ℓ. For this class, we first show a tight bound of $2\ell + 1$ under the SSYNC scheduler on the number of agents. We then prove the existence of a difference between FSYNC and SSYNC if the network size and the number of agents are known. In fact, in this case, while the bound for SSYNC remains unchanged, we prove a tight bound of 2ℓ for FSYNC. Moreover, we show that if $2\ell + 1$ agents are available in SSYNC, the exploration with termination is possible. As a corollary of these results, we re-establish a recently published bound for temporally-connected rings [5] and one for 1-interval connected rings [12].

Note that, when considering the class $\mathcal{H}(\ell)$ of temporally connected graphs with at most ℓ transient edges and the class $\mathcal{W}(\ell) \subset \mathcal{H}(\ell)$ of ℓ-bounded 1-interval connected graph, we have that the bound on the number of agents for $\mathcal{H}(\ell)$ is the same as the one for $\mathcal{W}(\ell)$ for SSYNC, while the two differs in the case of FSYNC, showing that the stronger connectivity assumption of $\mathcal{W}$ does not influence the solvability bound in case of semi-synchronous schedulers, but does have an impact for fully synchronous ones.

2 The Model

2.1 The Network

The system is modeled as a time-varying graph (TVG), $\mathcal{G} = (V, E, \mathbb{T}, \rho)$, where V is a set of nodes, E is a set of edges, $\mathbb{T}$ is the temporal domain, and $\rho : E \times \mathbb{T} \to \{0, 1\}$, called *presence function*, indicates whether a given edge is available at a given time. The graph $G = (V, E)$ is called *underlying* graph (or *footprint*) of $\mathcal{G}$, with $|V| = n$ and $|E| = m$. Let $E(v)$ denote the set of edges incident on node v in the footprint, let $\delta_v = |E(v)|$ be the degree of node v in the footprint, and let $\Delta = Max_v\{\delta_v\}$ be the maximum degree of G.

In this paper we consider *discrete* time; that is, $\mathbb{T} = \mathbb{Z}^+$. Since time is discrete, the dynamics of the system can be viewed also in terms of a sequence of static graphs: $\mathcal{S}_\mathcal{G} = G_0, G_1, \ldots, G_t, \ldots$, where $G_t = (V_t, E_t)$ is the graph of the edges present at time t (also called *snapshot* at time t). The TVG in this case is called *temporal graph* (or *evolving graph*). We denote by $\bar{E}_t = E \setminus E_t$ ($\subseteq E$) the set of edges that do not appear in the snapshot at time t.

In a temporal graph, the edge set E can be partitioned into the set of recurrent edges E^*, and the one of transient edges E^-. Formally, a *recurrent edge* $e^* \in E^*$ is such that $\forall t \in \mathbb{Z}^+, \exists t' > t : \rho(e^*, t') = 1$. In other words, a recurrent edge appears infinitely often. On the other hand, a *transient edge* $e^- \in E^-$ is such that $\exists t \in \mathbb{Z}^+, \forall t' \geq t : \rho(e^-, t') = 0$. In other words, a transient edge eventually ceases to exist forever.

The *solidity* of $\mathcal{G}$ is defined as the number $\sigma(\mathcal{G})$ of recurrent edges, and the *evanescence* of $\mathcal{G}$, denoted by $\eta(\mathcal{G})$, as the number of transient edges (i.e., $\eta(\mathcal{G}) = |E| - \sigma(\mathcal{G})$).

A *journey* is a temporal walk in $\mathcal{G}$ and it is defined as a sequence of couples $\mathcal{J} = \{(e_1, t_1), (e_2, t_2) \ldots, (e_k, t_k)\}$, such that $\{e_1, e_2, ..., e_k\}$ is a walk in G and $\forall i, 1 \leq i < k$, $\rho(e_i, t_i) = 1$ and $t_{i+1} > t_i$. Let $J(u, v, t)$ denote the set of journeys from u to v starting at time $t' \geq t$.

A particularly important class of temporal graphs are *temporally connected* ones:

▶ **Definition 1** (Temporally connected). *A TVG $\mathcal{G}$ is temporally connected (or connected over time) if $\forall t \in \mathbb{Z}^+$, $\forall u, v \in V$, $J(u, v, t) \neq \emptyset$.*

Note that temporal connectivity is the minimal condition to be able to perform any global tasks; in particular, perpetual exploration (i.e., requiring every node to be visited infinitely often) is trivially impossible if the graph is not temporally connected. Let $\mathcal{H}$ denote the class of temporally connected TVGs.

A variety of stronger assumptions have been studied in the literature. In this paper we are interested in a particular temporally connected graph, where connectivity is actually guaranteed at every time (*always connected* or *1-interval connected* temporal graphs); in particular, when the number of missing edges at any given time is bounded.

▶ **Definition 2** (ℓ-Bounded 1-Interval Connected). *A temporal graph $\mathcal{G}$ is 1-interval connected (or always connected) if $\forall G_i \in \mathcal{S}_\mathcal{G}$, G_i is connected. Moreover, $\mathcal{G}$ is ℓ-bounded 1-interval connected if it is always connected and $|\bar{E}_t| \leq \ell$.*

Let $\mathcal{W}(\ell) \subset \mathcal{H}$ denote the class of ℓ-bounded 1-interval connected temporal graphs.

The nodes of $\mathcal{G}$ are anonymous (i.e., they have no IDs) and each node provides a constant amount of local memory called *whiteboard*. Each edge incident to node v is locally labeled by a bijection $\lambda_v : E(v) \to \{0, \ldots, \delta_v - 1\}$; no other assumptions are made about the labels. Every node v has ports p_i for $0 \leq i \leq \delta_v - 1$ which are used to store at most one agent trying to move through e such that $\lambda_v(e) = i$.

2.2 Mobile agents

A set $A = \{a_0, a_1, \ldots, a_{k-1}\}$ of k agents operate on the network, initially occupying arbitrary positions. Agents are anonymous and have access to their private notebook (local memory) and to whiteboards (memory of nodes).

The agents operate in synchronous rounds, and each round is composed by three phases: Look, Compute, and Move, during which they execute the following actions [20]:

Look: Agent a_i observes the content of its own notebook and of the whiteboard of the node it occupies, and it checks, for each port of the node, if there are other agents at the same node.

COMPUTE: On the basis of the information obtained in the LOOK phase, a_i decides whether to move or not. It can write information on the whiteboard[1] and if it decides to move, it places itself in correspondence of the selected port (if it is not occupied by another agent).

MOVE: If a_i occupies a port, it tries to move. If the corresponding edge exists, a_i reaches the other side, otherwise it stays on the port. If a_i does not occupy a port, it does not move.

We distinguish between the *fully-synchronous* activation scheduler (FSYNC), when all the agents are activated in every round, and the *semi-synchronous* one (SSYNC), when an arbitrary subset of the agents is activated at each round. In SSYNC, the scheduler is an adversary which knows the algorithm of the agents, has infinite computing capacity, and tries to prevent agents from completing their task; however, it must activate every agent infinitely often. An agent which is not activated at round t is said to be *sleeping* at that round. The length of the sleeping time is finite but unbounded.

Under the semi-synchronous scheduler, we need to specify the behavior of the agents that fall asleep on a port when the corresponding edge is missing. In this paper, we assume the weakest rule, called *eventual transport rule* [12], in which the agent sleeping at a port will eventually be activated at a time when the edge corresponding to the port is present. This prevents the adversary from using semi-synchronicity to block an agent forever on a recurrent edge.

2.3 Configuration and execution

A configuration C_t is defined by: the contents of the whiteboards, the local memory of the agents, and the locations of the agents.An execution $\mathcal{E}^{\mathcal{A}} = C_0C_1\ldots$ of an algorithm $\mathcal{A}$ is an infinite sequence of configurations such that C_0 is an initial configuration (i.e., a configuration at round 0) and C_{t+1} is obtained from C_t by executing one round of algorithm $\mathcal{A}$. This execution is subject to two types of adversarial actions: those by the activation scheduler deciding which agents are activated in that round, and those of the topological scheduler deciding which edges are missing in that round. When no ambiguity arises, we use $\mathcal{E}$ instead of $\mathcal{E}^{\mathcal{A}}$.

2.4 The Exploration problem

We say that a node v is visited by round t if there exists a round t' $(0 \leq t' < t)$ such that an agent occupies v at time t'. We say that the network is explored by round t if every node has been visited by round t.

A *perpetual* exploration algorithm is one where, in every execution, every node is visited infinitely often. An exploration *with termination* algorithm is one where all the agents terminate after all nodes have been visited at least once. In this paper we are concerned with *perpetual exploration*.

3 Exploration of temporally connected TVGs

In this section, we show that the feasibility of exploration of temporally connected TVGs is related to their evanescence.

[1] Access to the whiteboard is done in fair mutual exclusion.

3.1 Impossibility

Let $\mathcal{H}(\ell) = \{\mathcal{G} \in \mathcal{H} : \eta(\mathcal{G}) \leq \ell\}$ be the class of temporally connected TVGs with evanescence at most ℓ. In this section we show that it is impossible to perform perpetual exploration of all $\mathcal{G} \in \mathcal{H}(\ell)$ with 2ℓ agents. The result is quite strong as it applies also to TVGs that are connected at every time step, with uniquely labeled nodes and agents, under a fully-synchronous scheduler, and in presence of topological knowledge.

▶ **Theorem 3.** *There exist temporally connected time-varying graphs $\mathcal{G} \in \mathcal{H}(\ell)$ that cannot be explored by $k = 2\ell$ agents. The result holds even if nodes and/or agents have distinct IDs, the network is always connected, the agents have some topological knowledge (n, m or k), and the scheduler is fully-synchronous.*

Proof. We show the theorem by constructing a graph $\mathcal{G} \in \mathcal{H}(\ell)$ that cannot be explored by 2ℓ agents by any algorithm. The main point of this proof is that an agent can eventually have only one of these two behaviors when wishing to traverse an edge that is missing: (i) the agent stays permanently on the chosen port, waiting for the appearance of the continuously missing edge; (ii) the agent eventually chooses a different edge. The former type of agents are called (with respect to the number of changes of a selected edge) *finite* and the latter *infinite*.

The components for constructing the graph are as follows. For $0 \leq i \leq 2\ell - 1$ $(= k - 1)$, let S_i^{inf} be a star with center node c_i^{inf} and 3 leaf nodes $\{b_{(i,0)}^{\mathsf{inf}}, b_{(i,1)}^{\mathsf{inf}}, b_{(i,2)}^{\mathsf{inf}}\}$ and S_i^{fin} be a star with center node c_i^{fin} and 3 leaf nodes $\{b_{(i,0)}^{\mathsf{fin}}, b_{(i,1)}^{\mathsf{fin}}, b_{(i,2)}^{\mathsf{fin}}\}$. We construct the graph using S_i^{inf}, S_i^{fin} and an additional node u.

Each component is connected as follows. For S_i^{inf} $(0 \leq i \leq 2\ell - 1)$ and u, each $b_{(i,j)}^{\mathsf{inf}}$ $(0 \leq j \leq 2)$ is connected with u by edge $(b_{(i,j)}^{\mathsf{inf}}, u)$. For S_i^{fin} $(0 \leq i \leq 2\ell - 1)$ and u, each $b_{(i,j)}^{\mathsf{fin}}$ $(j = 0$ or $1)$ is connected with u by edge $(b_{(i,j)}^{\mathsf{fin}}, u)$. In addition to that, for $0 \leq i \leq l - 1$, $b_{(2i,2)}^{\mathsf{fin}}$ and $b_{(2i+1,2)}^{\mathsf{fin}}$ are connected by $(b_{(2i,2)}^{\mathsf{fin}}, b_{(2i+1,2)}^{\mathsf{fin}})$. A graph for $l = 2$ $(k = 4)$ is depicted in Figure 1.

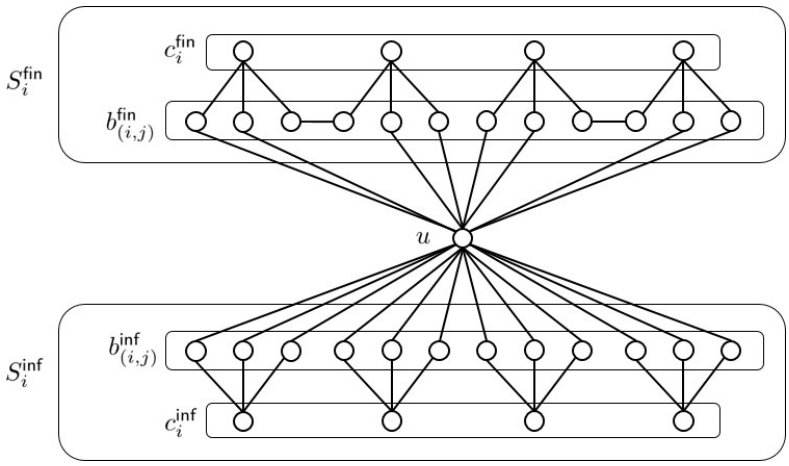


Figure 1 Example of a graph for $\ell = 2$ and $k = 2\ell = 4$. There are four stars S_i^{fin} (S_i^{inf}) for $0 \leq i \leq 3$ on the top (bottom) of the figure. Each star S_i^{fin} (S_i^{inf}) has one center node c_i^{fin} (c_i^{inf}) and three leaf nodes $\{b_{(i,0)}^{\mathsf{fin}}, b_{(i,1)}^{\mathsf{fin}}, b_{(i,2)}^{\mathsf{fin}}\}$ $(\{b_{(i,0)}^{\mathsf{inf}}, b_{(i,1)}^{\mathsf{inf}}, b_{(i,2)}^{\mathsf{inf}}\})$.

For the constructed graph, we first show that, given any exploration algorithm using 2ℓ agents, the adversary can construct an execution for the algorithm such that in the execution $\mathcal{G}$ cannot be explored while the adversary may violate the restriction of $\mathcal{H}(\ell)$, i.e., $\eta(\mathcal{G})$ may

be more than ℓ. Then, we give a way to convert the execution into another execution such that $\eta(\mathcal{G})$ is at most ℓ in the new execution and the agents cannot distinguish these two executions and thus cannot explore $\mathcal{G}$ also in the new execution.

We start by showing that, given any exploration algorithm, say $\mathcal{A}$, using 2ℓ agents, the adversary can construct an execution $\mathcal{E}_1$ of $\mathcal{A}$ in which the agents cannot explore $\mathcal{G}$. The adversary puts agent a_i on c_i^{inf} for $0 \leq i \leq 2\ell - 1$ in the initial configuration of $\mathcal{E}_1$. During execution $\mathcal{E}_1$ of $\mathcal{A}$, the adversary deletes edge $(b_{(i,j)}^{\mathsf{inf}}, u)$ whenever a_i is on $b_{(i,j)}^{\mathsf{inf}}$. Clearly, this prevents any agent executing $\mathcal{A}$ to visit u and thus $\mathcal{G}$ is not explored permanently while the adversary violates the restriction for the number of transient edges (it is at most 2ℓ in $\mathcal{E}_1$).

We now show how the adversary converts $\mathcal{E}_1$ into another execution, say $\mathcal{E}_2$, so that the agents cannot distinguish $\mathcal{E}_1$ and $\mathcal{E}_2$ and $\eta(\mathcal{G})$ is at most ℓ in $\mathcal{E}_2$. To decide the initial configuration of $\mathcal{E}_2$, the adversary first separates the agents into two groups: *finite agents* and *infinite agents* depending on their behavior when faced with a missing edge during $\mathcal{E}_1$. Let f $(0 \leq f \leq k-1)$ be the number of *finite agents*. In the following, *finite agents* are denoted by $a_0^{\mathsf{fin}}, \ldots, a_{f-1}^{\mathsf{fin}}$, and the *infinite agents* are denoted by $a_0^{\mathsf{inf}}, \ldots, a_{k-f-1}^{\mathsf{inf}}$. W.l.o.g., we assume that $a_i^{\mathsf{fin}} = a_i$, i.e, a_i^{fin} is the agent starting from c_i^{inf} in $\mathcal{E}_1$.

The adversary decides the initial configuration of $\mathcal{E}_2$ as follows: each a_i^{inf} $(0 \leq i \leq k-f-1)$ is put on the same node as in the initial configuration of $\mathcal{E}_1$, while each a_i^{fin} $(0 \leq i \leq f-1)$ is put on c_i^{fin}.

Then, the adversary changes the assignment of the port labels and the node ID (if any) of c_i^{fin}, $b_{(i,0)}^{\mathsf{fin}}$, $b_{(i,1)}^{\mathsf{fin}}$, and $b_{(i,2)}^{\mathsf{fin}}$ in S_i^{fin} so that a_i^{fin} cannot distinguish $\mathcal{E}_1$ and $\mathcal{E}_2$. Let $v_i = b_{(i,x)}^{\mathsf{inf}}$ be the node where $a_i = a_i^{\mathsf{fin}}$ finally waits a missing edge permanently in $\mathcal{E}_1$. For $b_{(i,2)}^{\mathsf{fin}}$, the assignment of the port labels and the node ID (if any) are copied from v_i. The ones of c_i^{fin} are copied from c_i^{inf}. The ones of $b_{(i,0)}^{\mathsf{fin}}$ and $b_{(i,1)}^{\mathsf{fin}}$ are copied from each of $b_{(i,y)}^{\mathsf{inf}}$ for $y \neq x$.

Execution $\mathcal{E}_2$ with the initial configuration, the node ID, and the assignment of port labels is constructed similarly to $\mathcal{E}_1$: the adversary deletes the edge leading to u (resp, u or $S_{i'}^{\mathsf{fin}}$ for $i' \neq i$) when $a_{i''}^{\mathsf{inf}} = a_i$ (resp, a_i^{fin}) exists on $b_{(i,j)}^{\mathsf{inf}}$ (resp, $b_{(i,j)}^{\mathsf{fin}}$). Obviously, every agent cannot distinguish $\mathcal{E}_1$ and $\mathcal{E}_2$: for all the agents, the node IDs and the port labeling observed in $\mathcal{E}_2$ is the same as $\mathcal{E}_1$. Thus, $\mathcal{G}$ cannot be explored since u is not visited by any agent also in $\mathcal{E}_2$.

Finally, we show that, in $\mathcal{E}_2$, $\eta(\mathcal{G})$ is at most ℓ. To prevent infinite agents, no transient edge is necessary; in fact, an infinite agent eventually changes its selected edge if it is kept missing, and no two infinite agents wait on the same edge (otherwise, the edge may be transient). For finite agents, by construction, a_{2i}^{fin} and a_{2i+1}^{fin} for $0 \leq i \leq (f-1)/2$ eventually wait for the same edge $(b_{(2i,2)}^{\mathsf{fin}}, b_{(2i+1,2)}^{\mathsf{fin}})$ (when f is odd, only a_{f-1} waits for $(b_{(f-1,2)}^{\mathsf{fin}}, b_{(f,2)}^{\mathsf{fin}})$). Since f is at most $k = 2\ell$, at most ℓ edges are necessary to prevent finite agents. ◀

3.2 Semi Synchronous Exploration by $2\eta(\mathcal{G}) + 1$ agents

In this section, we show that every temporally connected time-varying network $\mathcal{G} \in \mathcal{H}$ can be explored by $2\eta(\mathcal{G}) + 1$ anonymous agents that do not know the topology. In fact, we propose an exploration algorithm for $2\eta(\mathcal{G}) + 1$ anonymous agents in an anonymous network, which works under the semi-synchronous scheduler with eventual transport.

The strategy is simple and it is based on the classical *rotor router* mechanism, which was introduced as a deterministic alternative to random walk and was studied in a variety of contexts, including static graph exploration (e.g., [3, 29, 35]).

In rotor router, each node v has a variable written on its whiteboard, $\mathsf{pointer}_v$, indicating one of its incident ports. When an agent a visits node v, a checks each port in ascending

order from the port pointed by $\mathsf{pointer}_v$. If a finds some unoccupied port p, a moves to that port and sets $\mathsf{pointer}_v$ to $p+1$. If a finishes to check all the ports and they all are occupied, a does nothing.

Algorithm 1 Computation at node v.

```
if not on a port then
    i ← 0
    p ← pointer_v
    while i < δ_v ∧ port p is occupied do
        p ← (p + 1) mod δ_v
        i ← i + 1
    if i < δ_v then
        pointer_v ← (p + 1) mod δ_v
        move to port p
```

We first show that in any round, there exists at least one agent succeeding to move within finite time (Lemma 4). We then show that, $2l+1$ agents achieve perpetual exploration using Algorithm 1 (Theorem 5).

▶ **Lemma 4.** *For any round t, if $2\eta(\mathcal{G})+1$ agents execute Algorithm 1 in a temporally connected temporal graph $\mathcal{G}$, at least one of them eventually moves within finite time after t.*

Proof. By contradiction, assume that there exists a round t such that every agent never succeeds to move after t. We consider two cases: (i) there exists a node v containing more than $\delta_v - 1$ agents, and (ii) there does not exist such a node.

In the first case, every agent on v is activated within finite time after t because of the fairness of the scheduler, which means that every port of v is eventually occupied by an agent. Since at least one of the edges incident to v is a recurrent edge, say e, the agent sleeping on the corresponding port of e eventually succeeds to move because of the eventual transport rule. This is a contradiction.

Also in the second case, every agent on v is activated within finite time after round t because of the fairness of the scheduler. Since there is no node containing more agents than its degree, every agent eventually stays on a port. When this happens, at least one of the agents is sleeping at the port of a recurrent edge since the number of agents is $2\eta(\mathcal{G})+1$ and there exist at most $2\eta(\mathcal{G})$ ports corresponding to transient edges. This means that, by the eventual transport rule, the agent sleeping at the port of a recurrent edge eventually succeeds to move after t; a contradiction. ◀

Then, the following theorem holds.

▶ **Theorem 5.** *Any $\mathcal{G} \in \mathcal{H}$ can be explored by $2\eta(\mathcal{G})+1$ anonymous agents under the semi-synchronous scheduler.*

Proof. Consider Algorithm 1. By definition of transient edges, there exists a time step t_e such that, for any transient edge e, $\rho(e,t)=0$ for all $t > t_e$. Let t_E be $\max_{e\in E} t_e$, i.e., a time when all the transient edges have ceased to exist and all the edges that appear from this moment are recurrent. Let $x(t)$ be the sum of the number of visits over all the nodes from the beginning of the execution up to time t.

We now show that, from an arbitrary initial configuration, $2\eta(\mathcal{G})+1$ agents following Algorithm 1 visit all the nodes infinitely often.

First, note that there exists a node, say v, that is visited infinitely often (for $t \to \infty$) because $x(t)$ goes to infinity (for $t \to \infty$) by Lemma 4.

We now show that every neighbor of v connected by a recurrent edge is also visited infinitely often. We prove it by contradiction. Suppose that a neighbor u of v connected by a recurrent edge is visited only a finite number of times and let t' be the last round when u is visited. Since v is visited infinitely often and the agents execute Algorithm 1 perpetually, some agent a visiting v eventually chooses (v, u) as the edge from which a moves out of v after time t'. Recall that (v, u) is a recurrent edge and the agents are activated by the eventual transport rule. It follows that a eventually visits u after round t'; a contradiction.

Since G_r is temporally connected, we can apply inductively the claim (e.g., the neighbors of a neighbor of v is also visited infinitely often) to all the nodes, proving the theorem. ◀

From Theorems 3 and 5, the following Theorem holds.

▶ **Theorem 6.** *Exploration of all temporal graphs in $\mathcal{H}(\ell)$ is possible iff*

$k \geq 2\ell + 1$

Note that, if a graph is temporally connected, then its solidity $\sigma(\mathcal{G}) \geq n - 1$; as a consequence, we have:

▶ **Theorem 7.** *Every temporally connected temporal graph can be explored by $2(m - n) + 3$ agents.*

4 Exploration of 1-interval connected temporal graphs with bounded missing edges

In this Section, we turn our attention to the class $\mathcal{W}(\ell)$ of 1-interval connected temporal graphs where the number of missing edges is bounded in each round by a constant ℓ. In other words, at any time t the TVG is connected, and no more than ℓ edges are missing. We establish tight bounds for the exploration of this class of temporal graphs, in SSYNC and in FSYNC.

4.1 Semi-synchronous model

We first consider ℓ-bounded, 1-interval connected TVGs operating under a semi-synchronous scheduler and we show that there exists TVGs that cannot be explored by 2ℓ agents.

▶ **Theorem 8.** *There exist 1-interval connected time-varying graphs $\mathcal{G} \in \mathcal{W}(\ell)$ that cannot be explored by $k = 2\ell$ anonymous agents. The result holds even if the agents have some topological knowledge (n, m or k).*

Proof. We use the same graph $\mathcal{G}$ constructed for the proof of Theorem 3. The construction is omitted in this proof.

We first show that, given any exploration algorithm, say $\mathcal{A}$, using 2ℓ agents, the adversary can construct an execution $\mathcal{E}_1$ of $\mathcal{A}$, possibily violating the eventual transport rule, in which the agents cannot explore $\mathcal{G}$. We then show that it is always possible to convert this execution into another execution $\mathcal{E}_2$ that does not violate the eventual transport rule, and where the agents cannot explore $\mathcal{G}$.

In execution $\mathcal{E}_1$, the adversary puts agent a_i on c_i^{inf} for $0 \leq i \leq k - 1 = 2\ell - 1$ in initial configuration of $\mathcal{E}_1$. During $\mathcal{E}_1$, exactly one agent is activated at each round: a_i is activated at round t when $t \equiv i (\text{mod } k)$. When the adversary activates a_i and a_i exists on $b_{(i,j)}^{\mathsf{inf}}$, the

adversary deletes $(b^{\mathsf{inf}}_{(i,j)}, u)$ whereas all the other edges are present. Note that the agents and the nodes are anonymous and thus either they are all *finite* (i.e., every agent permanently waits for appearance of its selected edge if the edge is permanently missing) or they are all *infinite* (i.e., every agent eventually changes its selected edge if the edge remains missing) in $\mathcal{E}_1$. If the agents are *infinite*, the eventual transport rule is not violated even in $\mathcal{E}_1$ and thus the adversary can prevent the agents from completing the exploration in $\mathcal{E}_1$. If the agents are *finite*, the adversary converts $\mathcal{E}_1$ into another execution, say $\mathcal{E}_2$, as follows. The adversary first puts a_i $(0 \leq i \leq k-1)$ on c^{fin}_i in the initial configuration of $\mathcal{E}_2$. Then, the adversary changes the assignment of the port labels and the node ID (if any) of c^{fin}_i, $b^{\mathsf{fin}}_{(i,0)}$, $b^{\mathsf{fin}}_{(i,1)}$, and $b^{\mathsf{fin}}_{(i,2)}$ in the same way explained in the proof of Theorem 3 (also omitted in this proof). In $\mathcal{E}_2$, the adversary activates each agent in the same order as in $\mathcal{E}_1$ and deletes an edge leading to u or $S^{\mathsf{fin}}_{i'}$ for $i' \neq i$ whenever a_i is on $b^{\mathsf{fin}}_{(i,j)}$. After some round t from when every agent a_i does not change its selected edge at $b^{\mathsf{fin}}_{(i,2)}$ for $0 \leq i \leq 2l$, the adversary deletes $(b^{\mathsf{fin}}_{(2j,2)}, b^{\mathsf{fin}}_{(2j+1,2)})$ for $0 \leq j \leq l-1$ at every round. Obviously, every agent cannot distinguish $\mathcal{E}_2$ from $\mathcal{E}_1$ and $\mathcal{G}$ cannot be explored since u is not visited by any agent in $\mathcal{E}_2$. It is also clear that the eventually transport rule is not violated in $\mathcal{E}_2$. ◀

Clearly, $\mathcal{W}(\ell) \subset H(\ell)$, thus any $\mathcal{G} \in \mathcal{W}(\ell)$ can be explored by Algorithm 1; that is:

▶ **Theorem 9.** *Any $\mathcal{G} \in \mathcal{W}(\ell)$ can be explored by $2\ell + 1$ anonymous agents under the semi-synchronous scheduler.*

From Theorems 8 and 9 it follows that:

▶ **Theorem 10.** *Under a semi-syncrhonous scheduler, exploration of all ℓ-bounded 1-interval connected TVG $\mathcal{G}$ is possible iff*

$$k \geq 2\ell + 1$$

4.2 Fully-synchronous model

In this section, we show that, if the network size and the number of agents are known, there exists a difference between Fsync and Ssync in the exploration of ℓ-boundend 1-interval TVGs. In fact, we show that, $\mathcal{G} \in \mathcal{W}(\ell)$ can be explored if $k \geq 2\ell$, while there exist graphs that cannot be explored with $2\ell - 1$ agents.

4.2.1 Impossibility

We now consider ℓ-bounded, 1-interval connected TVGs operating under a fully-synchronous scheduler and we show that there exists TVGs that cannot be explored by $2\ell - 1$ agents, even if the agents know n, m, and k.

▶ **Theorem 11.** *There exist ℓ-bounded 1-interval time-varying graphs $\mathcal{G} \in \mathcal{W}(\ell)$ that cannot be explored by $k = 2\ell - 1$ anonymous agents in* Fsync. *The result holds even if the agents have some topological knowledge (n, m, k).*

Proof. Let $K_{2\ell} = (V_{2\ell}, E_{2\ell})$ be the complete graph with 2ℓ nodes where $V_{2\ell} = \{v_0, v_1, \ldots, v_{2\ell-1}\}$. It is well known that the edges of $K_{2\ell}$ can be colored with $2\ell - 1$ colors, that is, $E_{2\ell}$ can be partitioned into $2\ell - 1$ disjoint independent edge sets (or complete matchings): $E^{(0)}_{2\ell}, E^{(1)}_{2\ell}, \ldots, E^{(2\ell-2)}_{2\ell}$. For example, the following separation leads to disjoint independent edge sets: each $E^{(i)}_{2\ell}$ has ℓ edges, $(v_i, v_{2\ell-1})$, (v_{i-1}, v_{i+1}), (v_{i-2}, v_{i+2}), $\ldots$, (v_{i-l+1}, v_{i+l-1}), see Figure 2 (for simplicity, mod 2ℓ is omitted).

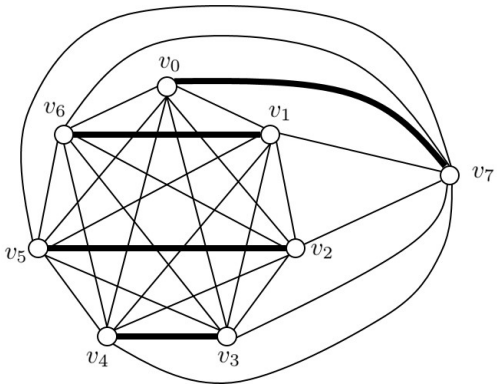


Figure 2 Example of coloring for the proof of Lemma 11. The bold lines are the edges of $E_8^{(0)}$.

The execution where v_{2l-1} remains unvisited is constructed as follows. For $0 \leq i \leq 2\ell - 1$, the adversary places each agent a_i on v_i and for $0 \leq j \leq 2\ell - 2$ assigns a label j to the port of v_i corresponding to e, if $e \in E_{2\ell}^{(j)}$. Note that, since agents and nodes are anonymous, all the agents select the port with the same label to move at each round. Thus, the adversary can prevent any agent from moving by deleting all the edges of $E_{2\ell}^{(i)}$ when the agent selects port i; as a consequence, none of the agents can move out of their current nodes. This means that $v_{2\ell-1}$ remains unvisited forever.

In this execution, the number of missing edges is always ℓ and the network is obviously kept connected. Thus, the theorem holds. ◀

4.2.2 Bound on Exploration time

Let $\mathcal{G} \in \mathcal{W}(\ell)$. Since $\mathcal{W}(\ell) \subset H(\ell)$, we can clearly execute Algorithm 1 in graph $\mathcal{G}$. Interestingly, when executed on $\mathcal{G} \in \mathcal{W}(\ell)$, it can be shown that the time complexity of exploration can be bounded under the fully-synchronous scheduler. More specifically, we show that within $\Delta^n(\Delta+1)^k(n-1)^k$ rounds, all nodes of the graph have been visited at least once by a team of $k = 2\ell + 1$ agents.

We prove the theorem by a sequence of lemmas. First of all, we can easily show that $2\ell + 1$ agents executing Algorithm 1 cannot be all prevented from moving at any given round.

▶ **Lemma 12.** *If $2\ell+1$ agents activated fully-synchronously execute Algorithm 1 in ℓ-bounded 1-interval TVGs, at least one of them succeeds to move at every round.*

Proof. There exist two cases as in the proof of Lemma 4: at round t, (i) there exists a node v containing more than $\delta_v - 1$ agents, and (ii) there does not exist such a node.

In the first case, since there are more than $\delta_v - 1$ agents at v, every port is occupied by one agent at t since every agent is activated. In addition to that, v has at least one adjacent edge present at t by the connectivity of the TVG. This implies that at least one agent succeeds to move at round t.

In the second case, each agent occupies one port by assumption and by fully-synchronous activation, which means that $2\ell + 1$ ports are occupied. Moreover, at most ℓ edges are missing at each round, which means that at most 2ℓ ports are blocked at each round. It follows that at least one agent can move at round t also in this case. ◀

To show the upper-bound on time complexity, we introduce the notions of *augmented configuration* and *augmented execution*.

In an augmented configuration C_t^{aug}, a new variable $\mathsf{visited}_v$ written and read only by an external observer, is added to each node v. The initial value of $\mathsf{visited}_v$ is 0. When v is visited, $\mathsf{visited}_v$ is set to 1 by the external observer. An augmented configuration C_t^{aug} is defined by configuration C_t and the value of $\mathsf{visited}_v$ of every node v at round t. We say that an augmented configuration is *terminal* when $\mathsf{visited}_v = 1$ for any node v.

An augmented execution $\mathcal{E}^{\mathsf{aug}} = C_0^{\mathsf{aug}} C_1^{\mathsf{aug}} \dots C_r^{\mathsf{aug}}$ is a sequence of augmented configurations such that C_0^{aug} is an initial augmented configuration; C_{t+1}^{aug} is obtained from C_t^{aug} by $2\ell+1$ agents executing one round of Algorithm 1 fully-synchronously, with the action of the adversary deciding which edges are missing; C_r^{aug} is a unique terminal configuration in $\mathcal{E}^{\mathsf{aug}}$. Note that the agents keep executing Algorithm 1 after round r, but augmented configurations after round r are ignored in $\mathcal{E}^{\mathsf{aug}}$. For $\mathcal{E}^{\mathsf{aug}}$, the following lemma holds.

▶ **Lemma 13.** *In an augmented execution by $2\ell+1$ agents, any two augmented configurations are different.*

Proof. First note that Lemma 12 precludes the same two consecutive augmented configurations C_t^{aug} and C_{t+1}^{aug} in an augmented execution where no agents move between C_t^{aug} and C_{t+1}^{aug}. Suppose that there exist two augmented configurations C_t^{aug} and $C_{t'}^{\mathsf{aug}}$ for $t < t'$ in an augmented execution $\mathcal{E}^{\mathsf{aug}}$. Let $\mathcal{E}_{t,t'}^{\mathsf{aug}} = C_t^{\mathsf{aug}} C_{t+1}^{\mathsf{aug}} \cdots C_{t'-1}^{\mathsf{aug}}$ be a subsequence of $\mathcal{E}^{\mathsf{aug}}$. In this case, the adversary can create an infinite augmented execution from $\mathcal{E}^{\mathsf{aug}}$ by repeating $\mathcal{E}_{t,t'}^{\mathsf{aug}}$, which means that the adversary can create an (augmented) execution where $2\ell+1$ agents cannot complete the exploration forever. This contradicts Theorem 5. Thus, the lemma holds. ◀

We are now ready to show an upper bound on the exploration time of Algorithm 1, which is obtained by calculating the maximum length among all the augmented executions.

▶ **Lemma 14.** *The length of any possible augmented execution by $k = 2\ell+1$ agents is bounded by $\Delta^n(\Delta+1)^k(n-1)^k$.*

Proof. Let α be the maximum length among all the possible augmented executions. By Lemma 13, α is bounded by the number of possible augmented configurations in an execution.

The number of possible configurations on a fixed node set $V' \subseteq V$ is bounded by $\Delta^{|V'|}(|V'|(\Delta+1))^k$, which corresponds to all the combinations of the directions of pointers (i.e., $\Delta^{|V'|}$) and all of the the agents' locations (i.e., $(|V'|(\Delta+1))^k$). Notice that only $\mathsf{pointer}_v$ of each node v is used as a variable in Algorithm 1. Since the number of visited nodes is not decreasing during the exploration, the exploration time is smaller than or equal to the sum of $\Delta^{|V'|}(|V'|(\Delta+1))^k$ for $1 \le |V'| \le n-1$, i.e., $\alpha \le \Sigma_{|V'|=1}^{n-1} \Delta^{|V'|}(|V'|(\Delta+1))^k \le \Delta^n(\Delta+1)^k(n-1)^k$ rounds. ◀

It then follows that:

▶ **Theorem 15.** *Under a fully-synchronous scheduler, Algorithm 1 executed by $k = 2\ell+1$ anonymous agents explores any ℓ-bounded 1-interval connected TVG within $\Delta^n(\Delta+1)^k(n-1)^k$ rounds.*

Note that, as a consequence, we obtain a terminating exploration algorithm for ℓ-bouned 1-interval connected TVGs.

▶ **Theorem 16.** *With knowledge of n and k, exploration with termination of an arbitrary ℓ-bounded 1-interval connected temporal graph $\mathcal{W}(\ell)$ can be achieved in $\Delta^n(\Delta+1)^{2\ell+1}(n-1)^{2\ell+1}$ rounds by $2\ell+1$ agents under the fully-synchronous scheduler.*

4.2.3 Exploration by 2ℓ agents

The result of the previous section can be used to obtain a perpetual exploration algorithm of ℓ-bounded 1-interval connected graphs by 2ℓ agents (which know n and k). The solution (Algorithm 2 below) is obtained by applying Algorithm 1 bounding the waiting time of an agent blocked on a missing edge.

In fact, while an agent keeps waiting for a missing edge forever in Algorithm 1, in Algorithm 2 an agent waits for a missing edge up to kT rounds where T is calculated on the basis of the results of Section 4.2.2.

Apart from the waiting time, the rest of the algorithm is the same as in Algorithm 1: each node has $\mathsf{pointer}_v$ pointing to a port. When a visits v, a checks each port in ascending order from the port pointed by $\mathsf{pointer}_v$. If a finds some unoccupied port p, a moves to the port and sets $\mathsf{pointer}_v$ to $p+1$. If a finishes to check all the ports and they all are occupied, a does nothing.

Variable Waiting of an agent represents the elapsed time since the last round when the agent moved to the port.

Algorithm 2 Computation at node v

```
if on a port then
    Waiting ← Waiting + 1
    if Waiting > kT then
        exit the current port
if not on a port then
    Waiting ← 0
    i ← 0
    p ← pointer_v
    while i < δ_v ∧ port p is occupied do
        p ← (p + 1) mod δ_v
        i ← i + 1
    if i < δ_v then
        pointer_v ← (p + 1) mod δ_v
        move to the port p
```

▶ **Lemma 17.** *Let 2ℓ agents execute Algorithm 2. If an agent waits at u for a missing edge $e = (u, v)$ for kT rounds, during this time either another agent starts to wait for e at v, or the other $2\ell - 1$ agents complete the exploration.*

Proof. Suppose that an agent a at u starts to wait for a missing edge (u, v) at round t and (u, v) is kept missing for the next kT rounds (including t).

We first show that there exist T successive rounds in $[t, t+kT)$ during which all the agents but a do not satisfy predicate $\mathsf{Waiting} > kT$ even if their selected edge remains missing.

We show the claim by contradiction. We assume that in any interval of T successive rounds in $[t, t + kT)$, there is an agent that satisfies $\mathsf{Waiting} > kT$.

By assumption, at least k agents other than a must satisfy $\mathsf{Waiting} > kT$, since $kT/T = k$. This means that at least one agent (different from a) satisfies the predicate twice since the number of the agents (excluding a) is $k-1$. However, once an agent satisfies $\mathsf{Waiting} > kT$ at round $t' \in [t, t + kT)$, the agent never satisfies the predicate in $[t, t + kT)$ since the length of the interval is kT. This is a contradiction. Thus, there exist T successive rounds in $[t, t+kT)$ during which all the agents (except for a) do not satisfy $\mathsf{Waiting} > kT$ even if their chosen edge is kept missing.

Now, we show the lemma, i.e., show that another agent at v starts to wait for $e = (u, v)$ or the exploration is completed. Suppose that no agent at v starts to wait for e in these T rounds. Since e is missing during these T rounds, during that time the network (without e) can be considered as a $(\ell - 1)$-bounded 1-interval connected TVG. By Theorem 15, $2(l - 1) + 1 = 2\ell - 1$ agents complete the exploration of the $(\ell - 1)$-bounded TVGs in these T rounds. This means that the $2\ell - 1$ agents other than a complete the exploration of the network without e during those T rounds, because none of them starts to wait for e at v during that time by assumption. Thus, the lemma holds. ◀

▶ **Theorem 18.** *Any ℓ-bounded 1-interval connected temporal graph $\mathcal{G} \in \mathcal{W}(\ell)$ can be explored by $k = 2\ell$ anonymous agents with knowledge of n and k, under a fully-synchronous scheduler.*

Proof. The proof follows the same lines of Theorem 5. We first show that, executing Algorithm 2 , there exists at least one node v which is visited infinitely often, and we then show that all the nodes are visited infinitely often. Let $x(t)$ be the sum of the number of visits over all the nodes from the beginning of the execution up to time t and $V_A^{(t)}$ be a node set such that there exists at least one agent on every $w \in V_A^{(t)}$ at round t.

We show that $x(t)$ goes to infinity (for $t \to \infty$), which leads to the existence of a node v visited infinitely often. We consider the configuration at round t and show that after t, $x(t)$ eventually increases. Two cases are considered: *Case 1)* there exists a node $\hat{v} \in V_A^{(t)}$ with δ_v or more agents and *Case 2)* there does not exist such a node.

Case 1) Suppose that there exists a node $\hat{v}$ with $\delta_{\hat{v}}$ or more agents at round t. Note that at least one of the edges incident to $\hat{v}$ exists at round t because the network is 1-interval connected. In this case, at least one of the agents on $\hat{v}$ succeeds to move because all the ports of $\hat{v}$ are occupied. Therefore, $x(t)$ increases.

Case 2) Suppose that there does not exist a node $\hat{v}$ with $\delta_{\hat{v}}$ or more agents. We show that $x(t)$ increases within finite rounds from t by contradiction. We assume that no agent moves out of its current node after t. Clearly, there exists a node $\tilde{v} \in V_A^{(t)}$ which has a neighbor $\tilde{u}$ with no agent (otherwise, the exploration would have been completed). An agent changes its port if it is blocked by the same missing edge for kT rounds by Algorithm 2; an agent $\tilde{a}$ on $\tilde{v}$ eventually chooses $(\tilde{v}, \tilde{u})$ to move from $\tilde{v}$. At this round, the adversary must prevent $\tilde{a}$ from moving by deleting $(\tilde{v}, \tilde{u})$. This means that the adversary must prevent $2(\ell - 1) + 1 = 2\ell - 1$ other agents from moving by deleting $\ell - 1$ edges, which is impossible. This leads to a contradiction. Therefore, $x(t)$ increases and goes to infinity for $t \to \infty$, and thus a node (say v) visited infinitely often exists.

We now show that all the neighbors of v are also visited by agents infinitely often. We prove it by contradiction. Suppose that a neighbor u of v is visited only a finite number of times and let t' be the last round when u is visited. Since v is visited infinitely often and the agents execute Algorithm 2, some agent a visiting v eventually chooses (v, u) as the edge from which a moves after t'. If (v, u) appears by the kT-th round since a chose it, a visits u as soon as (v, u) appears. Otherwise, another agent visits u by Lemma 17. It follows that u is eventually visited after t' rounds, which is a contradiction.

By the connectivity assumption, we can apply inductively the claim (e.g., the neighbors of a neighbor of v are also visited infinitely often) to all the nodes, proving the theorem. ◀

From Theorems 11 and 18, we have:

▶ **Theorem 19.** *Under the fully-synchronous scheduler, with knowledge of n and k, the exploration of all ℓ-bounded 1-interval connected TVGs is possible iff $k \geq 2\ell$.*

5 Conclusion

In this paper, we considered perpetual exploration of temporal graphs with arbitrary topology, focusing on the number of agents that are necessary and sufficient to perform the task. We considered two common dynamic models: temporally connected networks, and 1-interval connected (or always connected) networks with a bounded number of missing edges at each round. We derived tight bounds for both models under fully synchronous and semi-synchronous settings.

This is the first study on distributed exploration of temporal graphs with *arbitrary topology* and it has considered only temporally connected and 1-interval connected networks: the investigation of other connectivity classes of temporal graphs with arbitrary topology is the main research direction left open.

In this paper the focus was exclusively on feasibility of exploration; clearly, an important avenue of investigation is also the design of efficent solutions, whenever they exist.

References

1. S. Albers and M. Henzinger. Exploring unknown environments. *SIAM Journal on Computing*, 29(4):1164–1188, 2000.
2. C. Avin, M. Koucky, and Z. Lotker. How to explore a fast-changing world. In *Proc. of the 35th Int. Colloquium on Automata, Languages and Programming (ICALP)*, pages 121–132, 2008.
3. E. Bampas, L. Gasieniec, N. Hanusse, D. Ilcinkas, R. Klasing, A. Kosowski, and T. Radzik. Robustness of the rotor-router mechanism. *Algorithmica*, 78(3):869–895, 2017.
4. M. Bournat, A.K. Datta, and S. Dubois. Self-stabilizing robots in highly dynamic environments. *Theoretical Computer Science*, 772:88–110, 2019.
5. M. Bournat, S. Dubois, and F. Petit. Computability of perpetual exploration in highly dynamic rings. In *Proc. IEEE 37th Int. Conference on Distributed Computing Systems (ICDCS)*, pages 794–804, 2017.
6. A. Casteigts, P. Flocchini, W. Quattrociocchi, and N. Santoro. Time-varying graphs and dynamic networks. *Int. Journal of Parallel, Emergent and Distributed Systems*, 27(5):387–408, 2012.
7. J. Chalopin, P. Flocchini, B. Mans, and N. Santoro. Network exploration by silent and oblivious robots. In *Proc. of 36th International Workshop on Graph Theoretic Concepts in Computer Science (WG)*, pages 208–219, 2010.
8. R. Cohen, P. Fraigniaud, D. Ilcinkas, A. Korman, and D. Peleg. Label-guided graph exploration by a finite automaton. *ACM Trans. Algorithms*, 4(4):1–18, 2008.
9. S. Das. *Graph Exploration with Mobile Agents.* Chapter 16 of [20], pages 403–422, 2019.
10. X. Deng and C. H. Papadimitriou. Exploring an unknown graph. *Journal of Graph Theory*, 32(3):265–297, 1999.
11. G.A. Di Luna. *Mobile Agents on Dynamic Graphs.* Chapter 20 of [20], pages 549–584, 2019.
12. G.A. Di Luna, S. Dobrev, P. Flocchini, and N. Santoro. Live exploration of dynamic rings. In *Proc. IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 570–579, 2016.
13. Y. Dieudonné and A. Pelc. Deterministic network exploration by anonymous silent agents with local traffic reports. *ACM Transactions on Algorithms*, 11(2):Article No. 10, 2014.
14. S. Dobrev, L. Narayanan, J. Opatrny, and D. Pankratov. Exploration of high-dimensional grids by finite automata. In *Proc. 46th Int. Colloquium on Automata, Languages, and Programming (ICALP)*, pages 1–16, 2019.
15. T. Erlebach, M. Hoffmann, and F. Kammer. On temporal graph exploration. In *Proc. of 42th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 444–455, 2015.

16 T. Erlebach and J. T. Spooner. Faster exploration of degree-bounded temporal graphs. In *Proc. of 43rd International Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 1–13, 2018.

17 A. Ferreira. Building a reference combinatorial model for MANETs. *IEEE Network*, 18(5):24–29, 2004.

18 P. Flocchini, M. Kellett, P. Mason, and N. Santoro. Searching for black holes in subways. *Theory of Computing Systems*, 50(1):158–184, 2012.

19 P. Flocchini, B. Mans, and N. Santoro. On the exploration of time-varying networks. *Theoretical Computer Science*, 469:53–68, 2013.

20 P. Flocchini, G. Prencipe, and N. Santoro (Eds). *Distributed Computing by Mobile Entities*. Springer, 2019.

21 P. Fraigniaud, D. Ilcinkas, G. Peer, A. Pelc, and D. Peleg. Graph exploration by a finite automaton. *Theoretical Computer Science*, 345(2–3):331–344, 2005.

22 P. Fraigniaud, D. Ilcinkas, and A. Pelc. Impact of memory size on graph exploration capability. *Discrete Applied Mathematics*, 156(12):2310–2319, 2008.

23 T. Gotoh, Y. Sudo, F. Ooshita, H. Kakugawa, and T. Masuzawa. Group Exploration of Dynamic Tori. In *Proc. IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 775–785, 2018.

24 B. Haeupler and F. Kuhn. Lower bounds on information dissemination in dynamic networks. In *Proc. 26th International Symposium on Distributed Computing (DISC)*, pages 166–180, 2012.

25 F. Harary and G. Gupta. Dynamic graph models. *Mathematical and Computer Modelling*, 25(7):79–88, 1997.

26 D. Ilcinkas, R. Klasing, and A.M. Wade. Exploration of constantly connected dynamic graphs based on cactuses. In *Proc. 21st International Colloquium Structural Information and Communication Complexity (SIROCCO)*, pages 250–262, 2014.

27 D. Ilcinkas and A.M. Wade. On the power of waiting when exploring public transportation systems. In *Proc. 15th International Conference on Principles of Distributed Systems (OPODIS)*, pages 451–464, 2011.

28 D. Ilcinkas and A.M. Wade. Exploration of the T-Interval-connected dynamic graphs: the case of the ring. *Theory Computing Systems*, 62(4):1144–1160, 2018.

29 A. Kosowski and D. Pajak. Does adding more agents make a difference? A case study of cover time for the rotor-router. *J. Comput. Syst. Sci.*, 106:80–93, 2019.

30 F. Kuhn, N. A. Lynch, and R. Oshman. Distributed computation in dynamic networks. In *Proc. 42nd ACM Symposium on Theory of Computing (STOC)*, pages 513–522, 2010.

31 F. Kuhn and R. Oshman. Coordinated consensus in dynamic networks. In *Proc. 30th Symposium on Principles of Distributed Computing (PODC)*, pages 1–10, 2011.

32 O. Michail and P. Spirakis. Traveling Salesman Problems in Temporal Graphs. *Theoretical Computer Science*, 634:1–23, 2016.

33 P. Panaite and A. Pelc. Exploring unknown undirected graphs. *Journal of Algorithms*, 33:281–295, 1999.

34 C. Shannon. Presentation of a maze-solving machine. In *Proc. of the 8th Conf. of the Josiah Macy Jr. Foundation (Cybernetics)*, pages 173–180, 1951.

35 V. Yanovsky, A. Bruckstein, and I. Wagner. A distributed ant algorithm for efficently patrolling a network. *Algorithmica*, 37(3):165–186, 2003.

Parallel and Distributed Algorithms for the Housing Allocation Problem

Xiong Zheng
Electrical and Computer Engineering Department, University of Texas at Austin, USA
zhengxiongtym@utexas.edu

Vijay K. Garg
Electrical and Computer Engineering Department, University of Texas at Austin, USA
garg@ece.utexas.edu

Abstract

We propose parallel and distributed algorithms for the *housing allocation* problem. In this problem, there is a set of agents and a set of houses. Each agent has a strict preference list for a subset of houses. We need to find a matching for agents to houses such that some criterion is optimized. One such criterion which has attracted much attention is *Pareto optimality*. A matching is Pareto optimal if no coalition of agents can be strictly better off by exchanging houses among themselves. We also study the *housing market* problem, a variant of the housing allocation problem, where each agent initially owns a house. In addition to Pareto optimality, we are also interested in finding the a matching in the *core* of a housing market. A matching is in the *core* if there is no coalition of agents that can be better off by breaking away from other agents and switching houses only among themselves in the initial allocation.

In the first part of this work, we show that computing a Pareto optimal matching of a house allocation is in **CC** and computing a matching in the core of a housing market is **CC**-hard, where **CC** is the class of problems logspace reducible to the comparator circuit value problem. Given a matching of agents to houses, we show that verifying whether it is Pareto optimal is in **NC**. We also show that verifying whether it is in the core can be done in **NC**. We then give an algorithm to show that computing a maximum cardinality *Pareto* optimal matching for the housing allocation problem is in $\mathbf{RNC}^2$ and quasi-$\mathbf{NC}^2$.

In the second part of this work, we present a distributed version of the top trading cycle algorithm for finding a matching in the *core* of a housing market. To that end, we first present two algorithms for finding all the disjoint cycles in a functional graph. The first algorithm is a *Las Vegas* algorithm which terminates in $O(\log l)$ rounds with high probability, where l is the length of the longest cycle. The second algorithm is a deterministic algorithm which terminates in $O(\log^* n \log l)$ rounds, where n is the number of nodes in the graph. Both algorithms work in the synchronous distributed model and use messages of size $O(\log n)$. By applying these two algorithms for finding cycles in a functional graph, we give the distributed top trading cycle algorithm which terminates in $O(n)$ rounds and requires $O(n^2)$ messages.

2012 ACM Subject Classification Theory of computation → Parallel algorithms

Keywords and phrases Parallel Algorithm, Distributed Algorithm, Housing Allocation, Housing Markets, Pareto optimality

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.23

Related Version A full version of the paper is available at `https://arxiv.org/abs/1905.03111`.

Funding This work was partially supported by NSF CSR-1563544,CNS-1812349, and Cullen Trust Professorship.

© Xiong Zheng and Vijay K. Garg;
licensed under Creative Commons License CC-BY
23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller; Article No. 23; pp. 23:1–23:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Matching is a fundamental problem in computer science with numerous applications. The *housing allocation* problem [10, 23, 1] is an instance of matching problem with one-sided preferences. In this problem, we need to allocate a set H of houses among a set A of agents and monetary compensations are not allowed. Each agent $a_i \in A$ ranks in order of preference a subset of H (the acceptable houses for a_i). The variant in which there is an initial endowment of houses to agents is known as the *housing market* problem [19, 18, 17]. For both the housing market and the housing allocation problem, we need to construct a matching of agents to houses such that it is optimal with respect to some criterion. One criterion usually considered is *Pareto optimality* [1, 3, 19]. A matching M is Pareto optimal if there is no other matching M' such that no agent strictly prefers M to M', and at least one agent strictly prefers M' to M. For example, a matching M is not Pareto optimal if a group of agents could improve by exchanging the houses that they are assigned to in M.

Possible applications of the housing allocation problem and the housing market problem include: assigning virtual machines to servers in cloud computers, allocating graduates to trainee positions, professors to offices, and students to roommates. Yuan [21] also describes a large-scale application of housing allocation in the allocation of families to government-subsidized housing in China. Also, the paper [13] describes applications of algorithms for the stable marriage problem for mapping clients to server clusters in a content delivery network in Akamai. When only one side preference is considered, housing allocation algorithms can be applied.

For the housing allocation problem, there is a simple greedy algorithm, known as the *serial dictatorship mechanism* [1] to compute a Pareto optimal matching. The *serial dictatorship mechanism* works as follows. Arbitrate a total ordering on all the agents. Let all agents pick their top choice of the remaining houses one by one following the total order. This algorithm is sequential and takes $O(n^2)$ computation steps. Also, it does not necessarily give a maximum cardinality *Pareto* optimal matching. The paper [3] studies the problem of finding a maximum cardinality Pareto optimal matching for the housing allocation problem in the sequential setting. Their algorithm first computes a maximum cardinality matching of the bipartite graph formed by agents and houses and then improves the matching to be Pareto optimal. Their algorithm runs in $O(\sqrt{n}m)$ sequential time, where n is the number of agents plus the number of houses, and m is the number of edges of the agent-house bipartite graph. They also show that any improvement to the complexity of their algorithm would imply an improved algorithm for finding a maximum matching in a bipartite graph.

For the housing market problem, Shapley and Scarf [19] prove that there exists at least one matching in the core of any housing market and present the well-known *top trading cycle* (TTC) mechanism, which they attribute to David Gale. This mechanism works by repeatedly finding the top preference cycles and exchanging houses along those cycles. It takes $O(n^2)$ sequential steps. Ma [12] shows that the TTC mechanism is the only individually rational, Pareto-efficient, and strategy-proof mechanism. Roth and Postlewaite [18] show that there is exactly one matching in the core for each housing market instance. Note that the matching obtained by the TTC mechanism is not only a *Pareto* optimal maching, but also the unique *core*.

The parallel complexity of both these problems has not been studied in the literature. The housing allocation problem is a variant of the stable marriage problem with only one sided preferences. The decision version of the stable marriage problem, i.e., given a pair of man and woman, to decide whether they are matched in the man-optimal stable matching, is **CC**-complete [14]. The **CC** class [14] is the set of problems logspace reducible to the

comparator circuit value problem. Currently, there are no efficient parallel algorithms for this class of problems. It is conjectured that **CC** is incomparable with **NC** [6, 14], the class of problems computable in polylog parallel time. In this work, we show that finding the matching in the core of a housing market is **CC**-hard, which can be taken as evidence that this problem is not parallelizable. Although finding the core is hard, we show that given a matching, it can be verified in **NC** whether it is the core. On the other hand, finding a Pareto optimal matching is easier than finding the core. We show that finding a maximum cardinality Pareto optimal matching can be done in $\mathbf{RNC}^2$ and quasi-$\mathbf{NC}^2$, where $\mathbf{RNC}^2$ represents the problems which have uniform circuits of polynomial size and $O(\log^2 n)$ depth and quasi-$\mathbf{NC}^2$ represents the problems which have uniform circuits of quasi-polynomial size $n^{O(\log n)}$, and $O(\log^2 n)$ depth.

In this paper, we also study the housing market problem in the distributed setting. Specifically, we give a symmetric distributed algorithm for the TTC mechanism. By symmetric, we mean that each agent performs the same role.

In summary, this paper makes the following contributions:

- We prove that computing the core of a housing market is **CC**-hard, by giving a logspace reduction from the lexicographically first maximal matching problem, which is a **CC**-complete problem, to the housing market problem.
- We show that computing a maximum cadinality Pareto optimal matching for the housing allocation problem is in $\mathbf{RNC}^2$ and quasi-$\mathbf{NC}^2$.
- We give a symmetric distributed TTC algorithm for computing the core of a housing market, which runs in $O(n)$ rounds and requires $O(n^2)$ messages.

The paper is organized as follows. Section 2 gives preliminaries for the housing allocation and the housing market problem. Section 3 studies the parallel complexity of the housing market problem. Section 4 presents a parallel algorithm for computing a maximum cardinality Pareto optimal matching for the housing allocation. Section 5 presents a distributed algorithm for computing the core of a housing market. Finally, Section 6 presents the conclusions and future work.

2 Preliminaries

The *housing allocation* problem deals with assigning indivisible houses to agents who have preferences over these houses. In general, a housing allocation instance (A, H, P) consists of

(1) a set of agents $A = \{a_1, a_2, ..., a_n\}$,
(2) a set of indivisible houses $H = \{h_1, h_2, ..., h_m\}$,
(3) a preference profile $P = \{\prec_{a_1}, \prec_{a_2}, ..., \prec_{a_n}\}$, where $\prec_{a_i}$ defines a strict preference of agent a_i on a subset of houses.

We restrict our attention to strict preference profiles where each agent defines a strict total order over a subset of houses. Let $N(i)$ denote the subset of acceptable houses for agent i. The goal of the housing allocation problem is to find a *Pareto* optimal matching of agents to houses. For a matching μ and an agent i, let $\mu(i)$ denotes the house matched to agent i. We use $h \prec_i h'$ to denote that agent i prefers house h to house h'. For two matchings μ and ν, $\mu \prec_i \nu$ denotes that agent i prefers $\mu(i)$ to $\nu(i)$. The definitions of *Pareto Domination* and *Pareto optimality* [19] are given as below.

▶ **Definition 1.** *(Pareto Domination). Suppose μ, ν are matchings. Then μ Pareto dominates ν if and only if*

(1) *$\mu \preceq_i \nu$ for all $i \in A$,*
(2) *$\mu \prec_j \nu$ for some $j \in A$.*

We use $\mu \prec \nu$ to denote that matching μ Pareto dominates matching ν.

▶ **Definition 2.** *(Pareto Optimality). Suppose μ is a matching. Then μ is Pareto optimal if and only if it is not Pareto dominated by any other matching ν.*

The *housing market* problem is a variant of the housing allocation problem, where there is an initial endowment of houses to agents and we have the same number of agents and houses. Let μ_0 be a matching denoting the initial endowment of houses to agents. Let (A, H, P, μ_0) denote an instance of a housing market. In the housing market problem, in addition to Pareto optimality, we also want a matching to be individually rational [2] defined as follows.

▶ **Definition 3.** *(Individually Rational). Suppose μ is a matching of agents to houses in a housing market. Then μ is individually rational if $\mu(a) \preceq_a \mu_0(a)$ for all $a \in A$.*

Individual rationality means an agent is willing to give up its initially assigned house only when it can get a better house. To define the *core* of a housing market, let us first define the concept of *coalition*. Informally, given a matching μ, a coalition w.r.t μ is a set of agents $A' \in A$ such that, by only switching houses within themselves, each agent in A' can get a house at least as good as its house in μ and at least one agent gets a strictly better house.

▶ **Definition 4.** *(Coalition). Given a housing market (A, H, P, μ_0) and a matching μ, a set of agents $A' \subseteq A$ form a coalition w.r.t μ if there exists a matching ν such that*
(1) $\nu(a) \in \{\mu_0(b) \mid b \in A'\},\ \forall a \in A'$
(2) $\nu(a) \preceq_a \mu(a)\ \forall a \in A'$
(3) $\exists\ a \in A'$ *such that* $\nu(a) \prec_a \mu(a)$
Condition (1) says that to get matching ν from μ_0, the agents in A' only switch houses within themselves. Condition (2) means that in matching ν each agent in A' is matched to a house at least as good as the house it gets matched to in μ. Condition (3) means that at least one agent is matched to a better house in matching ν. The *core* [19] of a housing market is defined as follows.

▶ **Definition 5.** *(Housing Market Core). The core of a housing market problem is a set of matchings M such that matching $\mu \in M$ if and only if there does not exist any coalition A' w.r.t μ.*

Essentially, a matching is in the core of a housing market if there does not exist a set of agents such that they can match to better houses by breaking away from other agents and exchanging houses within themselves. An individually rational and Pareto optimal matching is not necessarily a core matching, whereas a core matching must be individually rational and Pareto optimal. An example to illustrate the difference between a core matching and an individually rational Pareto optimal matching is given in Fig. 1.

$a_1 : h_2, h_3, h_1$	$a_1 : h_1$	$a_1 : h_2$	$a_1 : h_2$
$a_2 : h_1, h_3, h_2$	$a_2 : h_2$	$a_2 : h_3$	$a_2 : h_1$
$a_3 : h_1, h_2, h_3$	$a_3 : h_3$	$a_3 : h_1$	$a_3 : h_3$
Preference Profile	Initial Endowment	M_1	M_2

Figure 1 An Example.

It is easy to see that both M_1 and M_2 are individually rational and Pareto optimal matchings. M_2 is the core matching but M_1 is not. In M_1, agents a_1 and a_2 can form a coalition within themselves and swap houses. Suppose a_1 and a_2 break away from other agents and switch houses with each other. Then, a_1 gets house h_2, which is the same as the house it gets in M_1, and a_2 gets house h_1, which is strictly better than the house it gets in M_1. Thus, M_1 is not the core. On the other hand, in M_2, there does not exist two agents such that at least one will be strictly better off by forming a coalition and swapping houses among themselves.

The following result is well-known.

▶ **Lemma 6** ([18]). *There is exactly one unique matching in the core of a housing market instance.*

Since the core of a housing market has one unique matching, we use the core to mean this unique matching henceforth. The TTC algorithm given by Shapley and Scarf [19] computes the unique core of a housing market. This algorithm works in stages. At each stage, it has the following steps:

Step 1. Construct the *top choice* directed graph $G_t = (A, E)$ on the set of agents as follows. Add an arc from agent $i \in A$ to agent $j \in A$ if j holds the current top house of i.

Step 2. Since each node has exactly one outgoing edge in G_t, there must be at least one cycle, which could be a self-loop. All cycles are node disjoint. Find all the cycles in the top trading graph and implement the trade indicated by the cycles, i.e., each agent which is in any cycle gets its current top house.

Step 3. Remove all agents which get their current top houses and remove all houses which are assigned to some agent from the preference list of remaining agents.

The above steps are repeated until each agent is assigned a house. At each stage, at least one agent is assigned a final house. Thus, this algorithm takes $O(n)$ stages in the worse case and needs $O(n^2)$ computational steps.

3 Parallel Algorithms for Housing allocation and Housing Market

In this section, we study the parallel complexity of the housing allocation and housing market problem. The parallel computation model we use here is the CREW PRAM model [11]. First, we show that computing a Pareto optimal matching in a housing allocation is **CC** by reducing this problem to the lexicographically first maximal matching problem (LFMM), which is a **CC**-complete problem [14]. In the LFMM problem, we are given a graph $G(V, E, \prec)$ where $\prec$ denotes a total ordering on the edges. If $e_1 \prec e_2$, we say that e_1 precedes e_2. The total order $\prec$ allows us to regard a matching M as a sequence $S_M = (e_1, e_2, ...)$ of edges in ascending order, i.e., $j < k \implies e_j \prec e_k$. Given two matchings M and N, we say $M \prec N$ if S_M lexicographically precedes S_N. The relation $\prec$ defines a total order over all maximal matchings. The minimum element, M_{lex}, of this order is call the lex-first maximal matching of $G(V, E, \prec)$. We need to decide whether a given edge e is in the lex-first maximal matching of the graph. Then, we show that computing the unique core of a housing market is **CC**-hard, by giving a logspace reduction from the LFMM problem to the housing market problem. We say a problem is **CC**-hard if every problem in **CC** reduces to it.

▶ **Theorem 7.** *Computing a Pareto optimal matching for a housing allocation is in **CC**.*

Proof. We reduce the problem of computing a Pareto optimal matching to the LFMM problem. Given a housing allocation instance (A, H, P), we construct an agent-house bipartite graph $G = (A \cup H, E, \prec)$ where $\prec$ denotes a total ordering of edges. There is

an edge from an agent $u \in A$ to a house $v \in H$ if v is acceptable to u. For each agent u, let $r_u : H \to [|N(u)|]$ denote its rank function, i.e., $r_u(v)$ denotes the rank of house v at agent u's preference list. We assign an arbitrary unique ordering to all the agents, i.e., a one-to-one function $f : A \to [n]$, where $n = |A|$. For each edge (u, v), we associate the tuple $< f(u), r_u(v) >$ with it. We define the total order $\prec$ on edges as the lexicographical ordering of the tuples associated with them. Clearly, $\prec$ defines a total ordering on all edges. We claim that the lex-first maximal matching M_{lex} of this graph corresponds to a Pareto optimal matching. Suppose not, then there exists another maximal matching M' which dominates M. There must exist an agent $u \in A$ such that u prefers $M'(u)$ to $M(u)$. We have that $r_u(M'(u)) < r_u(M(u))$, which means that $(u, M'(u)) \prec (u, M(u))$, contradicting the fact that M is the lex-first maximal matching. ◀

▶ **Corollary 8.** *There is a $\tilde{O}(\sqrt{|E|})$ time parallel algorithm which uses $O(|E|)$ processors to compute a Pareto optimal matching, where $|E|$ is the number of acceptable agent-house pairs.*

Proof. Follows from the fact that there is a $\tilde{O}(\sqrt{|E|})$ time parallel algorithm for the LFMM problem [14]. ◀

▶ **Corollary 9.** *There is a $O(\sqrt{n})$ round distributed algorithm in the congest clique model [8] for computing a Pareto optimal matching, where n is the number of agents and houses.*

Proof. The paper [4] gives a $O(\sqrt{n})$ distributed algorithm for the weighted stable marriage problem. The LFMM problem is simply a subcase of the weighted stable marriage problem. Thus, the same algorithm can be applied here. ◀

▶ Remark 10. In the housing market problem, an individually rational and Pareto optimal matching must be a perfect matching of houses to agents. Thus, the reduction given in Theorem 7 cannot be applied. Instead, the problem of computing a individually rational and Pareto optimal matching can be reduced to the problem of lex-first perfect matching. Unfortunately, the complexity of this problem is unknown.

In a housing market (A, H, P, μ_0), the weighted agent-house bipartite graph $G = (A \cup H, E, w)$ is defined as follows. There is an edge between agent $a_i \in A$ and house $h_j \in H$ if either $h_j = \mu_0(a_i)$ or a_i prefers h_j to $I(a_i)$. The weight of the edge is defined as the rank of h_j in a_i's preference list. To compute an individually rational and Pareto optimal matching for a housing market, we first observe the following lemma from [7].

▶ **Lemma 11** ([7]). *A minimum weight perfect matching of the weighted agent-house bipartite graph is an individually rational and Pareto optimal matching.*

Proof. Let u be a minimum perfecting matching of the agent-house bipartite graph. Suppose u is not Pareto optimal. Then there must be another perfect matching v such that v Pareto dominates u. By the definition of Pareto domination, we can easily argue that v has smaller weight than u, contradiction. ◀

Combining with the results from [9] and [16], we have the following result.

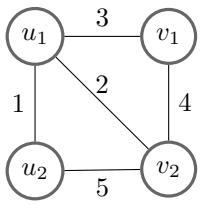
▶ **Theorem 12** ([7, 9, 16]). *There is a $\boldsymbol{RNC}^2$ algorithm and a quasi-$\boldsymbol{NC}^2$ algorithm for computing a individually rational and Pareto optimal matching of a housing market, which requires $O(n^3 \cdot m)$ and $n^{O(\log n)}$ parallel processors, respectively.*

Proof. By Lemma 11, to compute an individually rational and Pareto optimal matching, we can find a minimum weight perfect matching of the agent-house bipartite graph. By the results of [16] and [9], computing the minimum weight perfect matching of a bipartite graph is in $\mathbf{RNC}^2$ and quasi-$\mathbf{NC}^2$. ◀

Now we show that computing the core of a housing market is **CC**-hard.

▶ **Theorem 13.** *Computing the core of a housing market is **CC**-hard.*

Proof. We reduce the LFMM problem to the housing market problem. Let $G = (V, E, \prec)$ be an instance of a LFMM problem, where $\prec$ represents the total ordering on the edges. Let M_{lex} denote the lex-first maximal matching of G. We construct an instance for the housing market problem as follows. For each node $v \in V$, we create an agent a_v and a house h_v. So, we have $|A| = |H| = |V|$. Each agent a_v is initially assigned house h_v. The preference list for each agent is constructed based on the total ordering of edges in E. Note that to compute the core of a housing market, the preference list of an agent below its initial assigned house is irrelevant, since the core must be individually rational. So for each agent, we only need to specify the part of the preference list above its initial assigned house. For each pair of agents a_u and a_v, if edge (u, v) exists in graph G, then agent a_u prefers the house h_v of agent a_v to its own house h_u . Otherwise, agent a_u prefers its own house h_u to h_v. In other words, for each edge $(u, v) \in E$, agent a_u prefers house h_v of agent a_v to its own house h_u. The preference list of an agent a_u is defined based on the order of edges incident to vertex u, i.e., agent a_u prefers the house h_v of agent a_v to the house h_w of agent a_w if $(u, v) \prec (u, w)$. Since all edges are totally ordered, the preference list for each agent is strict. Fig. 2 shows an example of reduction above. Clearly, the above reduction can be done in logarithmic space.



LFMM Instance

$a_{u_1} : h_{u_2}, h_{v_2}, h_{v_1}, h_{u_1}$

$a_{u_2} : h_{u_1}, h_{v_2}, h_{u_2}, -$

$a_{v_1} : h_{u_1}, h_{v_2}, h_{v_1}, -$

$a_{v_2} : h_{u_1}, h_{v_1}, h_{u_2}, h_{v_2}$

Housing Market Instance

Figure 2 Constructing a Housing Market Instance from a LFMM Instance. At stage 0, edge (u_1, u_2) is added into M_{lex} by the greedy algorithm and vertices u_1 and u_2 are removed from the graph. In the TTC algorithm, the top choice graph only has one top trading cycle formed by agent a_{u_1} and a_{u_2}. Thus, agent a_{u_1} and a_{u_2} switch their houses and their houses are removed from the preference list of remaining agents. At stage 1, edge (v_1, v_2) is added into M_{lex} by the greedy algorithm. Agents a_{v_1} and a_{v_2} form a top trading cycle and switch houses with each other in the TTC algorithm.

We claim that an edge $e = (u, v) \in E$ is in M_{lex} if and only if agent a_u and a_v switch houses with each other in the core of the housing market instance. We say that an edge is minimum in its neighborhood if it is smaller than all its neighboring edges. Recall that the greedy algorithm for LFMM works as follows. Add each edge which is minimum in its neighborhood in the current graph into M_{lex} and remove all incident edges of the two endpoints of each such edge. Repeat the above procedure until the graph is empty. Since the greedy algorithm computes the unique lex-first maximal matching of a LFMM instance and the TTC algorithm computes the unique core of a housing market, it suffices to show that the TTC algorithm simulates the greedy algorithm on G.

Let $G^i = (V^i, E^i)$ denote the reduced graph at the beginning of stage i of the greedy algorithm. $G^0 = G$. Let R^i denote the set of edges in E^i added into M_{lex} by the greedy algorithm at stage i, i.e., the set of edges which are minimum in their neighborhoods in G^i. Let M^i_{lex} denote the set of edges in M_{lex} at the end of stage i. Let G^i_t denote the top choice graph formed by remaining agents at stage i of the TTC algorithm. We now show by induction on stages that an edge (u, v) is added into M_{lex} at stage i of the greedy algorithm iff the corresponding agents a_u and a_v switches houses at stage i of the TTC algorithm.

Base case: stage 0. Consider an edge $e = (u, v) \in R^0$. In the housing market, two agents a_u and a_v correspond to this edge. Since e is the minimum in its neighborhood, agent a_u and agent a_v are the top choice of each other. Thus, they form a top trading cycle of length 2 in G^0_t and switch houses with each other in the TTC algorithm. Therefore, all edges in R^0 correspond to the top trading cycles in G^0_t.

Induction case: assume the claim holds for stage i. Consider stage $i + 1$ of both algorithms. At the end of stage i, in the greedy algorithm, all edges incident to edges in R^i are removed from the graph. In the TTC algorithm, all houses involved in the top trading cycles are removed from the preference list of remaining agents. We claim for each edge $e = (u, v) \in R^{i+1}$, the two corresponding agents a_u and a_v in the housing market form a top trading cycle of length 2 in G^{i+1}_t. Suppose not. Let $e' = (u', v') \in R^{i+1}$ be an edge such that agent $a_{u'}$ and agent $a_{v'}$ do not form a top trading cycle in G^{i+1}_t. We must have that either house $h_{v'}$ is not the top choice of agent a_u or house $h_{u'}$ is not the top choice of agent $a_{v'}$ or both. Without loss of generality, assume house $h_{v'}$ is not the top choice of agent $a_{u'}$. We have two cases.

Case 1: house $h_{v'}$ is not available for agent $a_{u'}$. Then, agent $a_{v'}$ participates in a certain top trading cycle before stage $i + 1$. By induction assumption, this means that there exists one edge $e_{v'}$ incident to vertex v' which is added into M_{lex} at a stage before $i + 1$, contradicting the fact that edge (u', v') exists in E^{i+1}.

Case 2: house $h_{v'}$ is available for agent $a_{u'}$ but is not the current top choice for $a_{u'}$. Then, there exists another agent $a_{w'}$ such that agent $a_{u'}$ prefers the house $h_{w'}$ of $a_{w'}$ to the house of agent $a_{v'}$. The existence of agent $a_{w'}$ indicates that it is not involved in any top trading cycle before stage $i + 1$. By induction assumption, there does not exist any edge $e_{w'}$ in M^i_{lex} which is incident to vertex w'. Thus, we have $(u', w') \in E^{i+1}$. The fact that agent $a_{u'}$ prefers the house of agent $a_{w'}$ to the house of $a_{v'}$ indicates that $(u', w') \prec (u', v')$ which contradicts the fact that (u', v') is minimum in its neighborhood. ◀

Even though we do not know any **NC** algorithm for either computing an individually rational and Pareto optimal matching or computing the core of a housing market, given a matching, we can verify whether it is an individually rational and Pareto optimal matching and whether it is the core in **NC**.

▶ **Theorem 14.** *Given a matching μ of houses to agents in a housing market (A, H, P, μ_0), the following two tasks can be performed in **NC**.*

1) *Verifying whether μ is individually rational and Pareto optimal.*

2) *Verifying whether μ is the unique core.*

Proof.

1). Similar to verifying Pareto optimality in housing allocation [7], detailed proof given in the full paper [22].

2). Given a matching μ, to verify whether it is the core, we construct a different directed graph $G'(V', E')$ as follows. V' represents the set of agents. There are two types of arcs: solid arcs and dashed arcs. There is a solid arc from agent u to agent v if $\mu(u) = \mu_0(v)$,

i.e., agent u is assigned the house owned by agent v. Hence, the solid arcs represent how agents switch houses to get matching μ from the initial matching μ_0. Thus, all solid arcs form a set of disjoint directed cycles. There is a dashed arc from agent u to agent v if $\mu_0(v) \prec_u \mu(u)$.

We claim that μ is the core iff there is no directed cycle which contains dashed arcs in G. We show that any directed cycle with at least one dashed arc represents a coalition, w.r.t μ. Let C be such a cycle in G. If we switch houses following the cycle C, i.e., for each arc $< u, v >$ in C, agent u matches to house $\mu_0(v)$. For each solid arc $< u, v > \in C$, we have $\mu(u) = \mu_0(v)$ by the definition of solid arc. Thus, each agent with a solid outgoing arc matches to the same house as in μ. For each dashed arc $< u, v > \in C$, we have $\mu_0(v) \prec_u \mu(u)$, thus each agent with a dashed outgoing arc matches to a house strictly better than its house in μ. Thus, each directed cycle with at least one dashed edge represents a coalition of agents, w.r.t μ. Since a matching μ is in the core iff there is no coalition with respect to μ and each directed cycle with dashed arcs represent a coalition, we get our desired claim.

To check whether there exists a directed cycle with at least one dashed arc in G', we first compute the transitive closure TC' of G'. For each dashed arc $< u, v >$, check in parallel whether $TC'(v, u) = 1$. Thus verifying whether a matching is the core can be done in **NC**. ◀

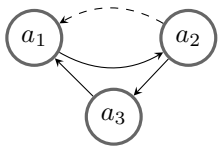


Figure 3 To verify whether matching M_1 in the example given in Fig. 1 is the core. The solid arcs represents how agents switch houses to get matching M_1. In M_1, a_2 prefers house h_1 which is the initial house of a_1, thus there is a dashed arc from a_2 to a_1. a_1 and a_2 form a coalition w.r.t M_1.

3.1 A Parallel Algorithm for Maximum Pareto Optimal Matching

In the housing allocation problem, a Pareto optimal matching does not necessarily have maximum cardinality, i.e., with maximum number of agents matched to a house. To find a maximum cardinality Pareto optimal matching, we adapt the sequential algorithm in [3] to be a parallel algorithm. The sequential algorithm in [3] has three steps. To ensure that the final matching has maximum cardinality, step 1 computes a maximum cardinality matching. After step 1, all unmatched agents are removed from consideration. At step 2, the algorithm improves the matching obtained from step 1 to be *trade-in-free*. A matching M is *trade-in-free* if there is no (agent,house) pair (a_i, h_j) such that a_i is matched in M, h_j is unmatched in M, and a_i prefers h_j to $M(a_i)$. That is, step 2 ensures that no matched agents prefers an unmatched house to its current matched house. After step 2, all unmatched houses are removed from consideration, since no matched agents prefer any of those houses to their matched houses. The final step is to improve the matching obtained from step 2 to be Pareto optimal, which is achieved by directly applying the TTC mechanism on all matched agents.

Our parallel algorithm, shown in Algorithm 1 has only two steps. At step 1, we compute the maximum cardinality matching, which can be reduced to compute a minimum weight perfect matching of a new graph. Let M' be the maximum cardinality matching obtained

at step 1. Let A' be the set of matched agents. After step 1, all the unmatched agents are removed from consideration. At step 2, we improve the matching obtained from step 1 to be Pareto optimal. In contrast to [3], we do not first make our matching *trade-in-free* and then *Coalition-free*. Instead, we directly compute a Pareto optimal matching by computing a minimum weight perfect matching of a graph G' constructed as follows. We create a set of virtual agents B' to ensure the number of agents is equal to the number of houses. Add an edge with weight 0 between each virtual agent and each house. For each real agent $a_i \in A'$ and each house $h_j \in H$, add an edge between a_i and h_j if h_j is a_i's partner at the end of step 1 or a_i prefers h_j to its partner. The weight of edge (a_i, h_j) is equal to the rank of h_j in a_i's preference list.

Algorithm 1 Pareto Optimal Matching for housing allocation.

Find a Maximum Cardinality Pareto Optimal Matching:

Step 1:
Let $G = (A \cup H, E)$ denote the agent-house bipartite graph.
Compute a maximum cardinality matching of G, denoted as M.

Step 2:
Let A' denote the set of matched agents in M.
Create a set of virtual agents B' such that $|A'| + |B'| = |H|$
Let E' denote the edge set
Add an edge with weight 0 into E' between each agent in B' and each house in H
forall $a_i \in A', h_j \in H$ **in parallel**:
  if $M(a_i) = h_j \vee M(a_i) \prec_{a_i} h_j$.
    $E' := E' \cup (a_i, h_j)$;
    $w'(a_i, h_j) :=$ rank of h_j in a_i's preference list
endfor
$G' = (A' \cup B' \cup H, E', w')$
Compute a minimum weight perfect matching of G', denoted as $M^{'}$
Output $M^* := \{(a_i, h_j) \in M^{'} \mid a_i \in A'\}$

Let $G'(A' \cup B' \cup H, E', w')$ be the graph constructed at Step 2. The following lemma shows the correctness of Algorithm 1.

▶ **Lemma 15.** *The matching output by Algorithm 1 is a maximum cardinality Pareto optimal matching for a housing allocation.*

Proof. Let M' be the minimum weight perfect matching of G'. Let M^* be the matching output by Algorithm 1, which is the induced submatching of M' on the set of matched agents A' after step 1. Step 1 ensures that M^* is a maximum cardinality matching. It remains to show that M^* is Pareto optimal. Suppose for contradiction that M^* is not Pareto optimal. Then there exists some other matching $M'' \neq M^*$ on real agents such that $M'' \prec M^*$. By definition of Pareto optimality, each agent in M'' should be matched to a house at least as good as the house in M^* and at least one agent is matched to a strictly better house in M''. Since M^* is a maximum cardinality matching, M'' must also be a maximum cardinality matching which matches the same set of agents as M^*. Since the weight of an edge (a_i, h_j) is defined as the rank of h_j at a_i's preference list, we have $w'(M'') < w'(M^*)$. Since each virtual agent has incident edges of weight 0 in G', there exists another perfect matching formed by edges in M'' and some edges incident to virtual agents such that the total weight is smaller than $M^{'}$, contradicting the fact that M' is the minimum perfect matching of G'. ◀

Now, we can state our main result for the housing allocation problem.

▶ **Theorem 16.** *There is a* $\boldsymbol{RNC}^2$ *and quasi-*$\boldsymbol{NC}^2$ *algorithm for finding a maximum cardinality Pareto optimal matching for the housing allocation problem.*

Proof. By Lemma 15, the matching obtained by Algorithm 1 is a maximum cardinality Pareto optimal matching. The time complexity of Algorithm 1 is dominated by the complexity of a minimum weight perfect matching of a graph. By [16] and [9], this step can be done in $\mathbf{RNC}^2$ and quasi-$\mathbf{NC}^2$. ◀

4 Distributed Algorithms for Housing markets

In this section, we present a symmetric distributed algorithm to implement the TTC mechanism in a distributed setting. We assume a distributed message passing model with n processes, $p_1, \ldots, p_n$, which form a completely connected topology. The system is synchronous, which means that there is an upper bound on the time for a message to reach its destination. We require that at each round, a node can only send a same message of $O(\log n)$ size to any other node in the network. Since the graph is fully connected, this model is also known as the congest clique model in the literature. Actually, our proposed distributed algorithm fits in a more restricted model called the *broadcast congest clique* model [8], since at each round each node only sends the *same* message to all other nodes in the network. This model is in contrast to the *unicast congest clique* model [8] which allows each node to send different messages to different nodes in each round.

To implement the TTC algorithm in a fully distributed way, we need efficient distributed algorithms for finding the top trading cycle. Observe that the graph formed by the top choice of each agent is a functional graph since each node has only one outgoing edge. Hence there is only one unique cycle in each connected component of this graph. We present two distributed algorithms for finding all the top trading cycles in a functional graph.

4.1 A Las Vegas Algorithm for Finding Cycles in Functional Graphs

In this section, we give a *Las Vegas* algorithm, shown in Algorithm 2, for finding all the disjoint cycles in a functional graph. The primary gradient of the algorithm is a pointer jumping technique. A similar technique is used in [15, 20] to solve the list ranking problem.

In this algorithm, each node has a variable *active*, which is initially true. A node terminates the code when *active* becomes false. Each node uses the variable *succ* to record its current successor node, which initially is its outgoing neighbor. Our algorithm will build a tree. The *children* variable denotes the current children of a node, which is essentially all the nodes that have been its successor. The *inCycle* variable denotes whether a node is in the cycle or not. The algorithm is composed of iterations and each node keeps executing an iteration until *active* becomes false. Each iteration includes two steps: a *Coin-flip* step and a *Explore* step. In the *Coin-flip* step, each active node flips a coin. If a node flips head and its successor node flips tail, it becomes inactive. This step is used to reduce the active nodes by a constant fraction. In the *Explore* step, each active node traverses along the path formed by the successor pointer of all nodes and tries to update its successor pointer to be the next active node in the path. It also adds all inactive node encountered into its *children* set. When such a active node is found, it checks whether such a node is actually itself, if that is the case, a cycle is detected. After a node determines that it is in the cycle (we will show that there is a unique such node), it broadcasts a cycle message along the tree formed by the child relationship (*Notify* step). We will prove that the set of nodes in the tree rooted at such

Algorithm 2 Randomized Algorithm for Finding the Cycles.

```
Code for P_i:
active := true
succ: successor of P_i //P_i's next active node
children: set of nodes that P_i traversed
inCycle: whether P_i is in the cycle, initially false

while active := true
    Coin-flip Step:
    Flip a coin, let myCoin denote the result
    Let succCoin be the coin result of succ
    if myCoin = head && succCoin = tail
        active := false
    Explore Step:
    if active := true
        Let succActive be the active status of succ
        while succActive = false
            children := children ∪ succ
            Let j be the successor of succ, set succ := j
            Let succActive be the active status of succ
        endwhile
        if succ = i /* now succ is also active /*
            active := false
endwhile

Notify Step:
if succ = i
    Send ("cycle") to children
On receiving ("cycle"):
    inCycle := true
    Send ("cycle") to all children
```

a node and formed by the child relation is exactly the set of cycle nodes. For the purpose of analysis, we assume the functional graph we consider only has one component, which also means it only has one cycle. Our algorithm works for functional graph with multiple components, since the executions on different components are independent.

We now show that at the end of Algorithm 2 each node correctly knows whether it is in the cycle or not. Let $succ[i]$ denote the value of *succ* for P_i. First, we can easily get the following lemma from the code.

▶ **Lemma 17.** *At the end of Algorithm 2, there is exactly one node i which has $succ[i] = i$ for each disjoint cycle in the functional graph.*

Proof. For node i, if $succ[i] = j$ at some point, then there exists a directed path from node i to node j. Let us consider a single connected component of the functional graph. For any non-cycle node, its *succ* cannot be itself since it does have a directed path to itself. Hence, it is sufficient to consider only cycle nodes. We first show there is at least one node i with $succ[i] = i$. We claim that there is at least one active cycle node remaining after the *Coin-flip* step of each iteration. To become inactive, a node has to flip head and its successor has to flip tail. This implies that two consecutive active nodes cannot become inactive simultaneously. Hence, at the end of the algorithm at least one cycle node i with $succ[i] = i$. Also, it is obvious that at most one cycle node i can have $succ[i] = i$ at the end of the algorithm. Therefore, there is exactly one node i which has $succ[i] = i$ for each disjoint cycle in the functional graph. ◀

Let i be the node with $succ[i] = i$ at the end of the algorithm. Let T be the tree rooted at node i and constructed from the *child* relation at the end of the algorithm. Let V_T denote the set of nodes in tree T. Let C denote the set of nodes in the cycle.

▶ **Lemma 18.** $C = V_T$

Proof. We prove $C \subseteq V_T$ and $V_T \subseteq C$. Suppose node i runs for L iterations. Let A_r denote the set of active cycle nodes at round r, $1 \leq r \leq L$. To prove $C \subseteq V_T$, we show by induction that each node in A_r is in tree T for all r.

Base case, $r = L$. $A_L = \{i\}$. Node i is the root of T.

Induction case: Suppose each node in A_k is in T. We need to show that each node in A_{k-1} is in T. It is sufficient to show that the nodes in A_{k-1} which become inactive at round k are in T. Since all active cycle nodes at each round still form a cycle, A_k divides A_{k-1} into multiple directed paths. For any path P of form $v_i, v_{i+1}, ..., v_j$, only the two end nodes v_i and v_j are in A_k. From the code we know that node v_i continues to find active nodes along P at round k, and it stops until it reaches node v_j. Thus, all nodes in path P between v_i and v_j become the children of node v_i. So, all nodes in path P are in T. Hence, all nodes in A_{k-1} are in T. Therefore, we have A_r is in T for any $1 \leq r \leq L$. Since A_1 is exactly the set of cycles nodes, we have all cycles nodes are in T. Thus, $C \subseteq V_T$.

To prove $V_T \subseteq C$, we show that the any non cycle node is not in tree T. For any non cycle node j, suppose $j \in V_T$. Then j must be a descendant of root node i. From the algorithm we know that the children relation is formed by next relation in the original graph. Thus, there must be a directed path from node i to node j in the original graph. This means j must be in the unique cycle, a contradiction. ◀

▶ **Theorem 19.** *Algorithm 2 computes all the cycles of a functional graph G. It has round complexity of $O(\log l)$ and message complexity of $O(n \log l)$, w.h.p, where l is the length of the longest cycle in G.*

Proof. Since the *cycle* message only traverses through tree T, from Lemma 18, we know that each cycle node receives the *cycle* message and each non cycle node does not receive the *cycle* message.

Since the number of active nodes in any cycle reduces by a constant fraction in expectation at each iteration and each iteration takes constant number of rounds, by Chernoff bound, Algorithm 2 takes $O(\log l)$ rounds w.h.p. Each round of the algorithm takes at most $O(n)$ messages, which results in $O(n \log l)$ messages in total. ◀

4.2 A Deterministic Algorithm for Finding Cycles in Functional Graphs

In this section, we present a deterministic algorithm for finding all the disjoint cycles in a functional graph, shown in Algorithm 3. This algorithm is similar to the *las vegas* algorithm in the previous section, with only one key difference. We replace the *Coin-flip* step in Algorithm 2 to the *Coloring* step. Observe that in Algorithm 2 the primary purpose of the *Coin-flip* step is to reduce the number of active nodes by a constant factor while ensuring that any two consecutive active cycle nodes cannot become inactive at the same time. Graph coloring techniques can also serve this purpose. Hence, we simply replace the *coin-flip* step in Algorithm 2 to be a *Coloring* step, which is an invocation of the 6-coloring algorithm due to [5]. After the *Coloring* step, each node compares its color with the color of its successor. If a node has a smaller color than its successor, it becomes inactive. Then all remaining active nodes perform the *Explore* step as in Algorithm 2.

We can observe that after the coloring step at each iteration, the node with the largest color remains active in each disjoint cycle. By similar argument, we can show that Lemma 17 and Lemma 18 still hold.

Algorithm 3 Deterministic Algorithm for Finding Cycles.

Code for P_i:
/* Variables are the same as Algorithm 2 */
while $active := true$
  Coloring Step:
  6-coloring of active nodes using coloring algorithm from [5]
  Request the color of $succ$, denoted as c'
  if $c < c'$ /* If my color is less than the color of successor, I become inactive */
    $active := false$
  Execute **Explore Step** of **Algorithm 2**
endwhile

▶ **Theorem 20.** *Algorithm 3 computes all the disjoint cycles in a functional graph and takes $O(\log^* n \log l)$ rounds and $O(n \log^* n \log l)$ messages.*

Proof. Since no more than 5 consecutive active nodes become inactive at each iteration by the property of 6-coloring, the *Explore Step* still takes constant rounds. The coloring step introduces an additional $O(\log^* n)$ factor. Thus, Algorithm 3 terminates in $O(\log^* n \log l)$ rounds and takes $O(n \log^* n \log l)$ messages. ◀

4.3 Distributed Top Trading Cycle Algorithm

We now present a distributed version of the top trading cycle algorithm. As in the sequential setting, we assume that each node knows which nodes are holding the houses in its preference list. Indeed, every node can broadcast its house to all. This only requires one round and $O(n^2)$ messages.

Algorithm 4 Distributed Version of Top Trading Cycle.

Code for P_i:
/* Variables */
$next_i$: the node which holds current top choice of P_i
$assigned_i$: whether be assigned final house
h_i: the house P_i holds
$succ_i$: successor of P_i, same as algorithm 2
$pref_i$: mapping from a house to the node which holds the house.

One Stage:
  $succ_i = next_i$
  Execute **Algorithm 2 or 3** to find out cycle nodes
  if P_i in cycle
    Let h_j denote the house of $next_i$
    $h_i := h_j$
    $assigned := true$
    Broadcast $remove(h_i)$ to all
    if P_i has no children
      Send ok to its parent

On receiving ok from all children:
  if P_i is the root of the tree
    Broadcast $nextStage$ to all
  else
    Send ok to its parent

On receiving $nextStage$:
  if $assigned_i := false$
    $active := true$
    Let Top_i denote the next available top choice
    $next_i := pref_i[Top_i]$
    Start next stage

On receiving $remove(h_j)$ from j:
  Remove h_j from $pref_i$

The distributed algorithm is shown in Algorithm 4. The basic idea is using the cycle finding algorithms presented above to simulate each stage of the top trading cycle algorithm. During a stage, all nodes first build the top choice functional graph, i.e., update their *succ* variable to be the node which holds their current top choice. Then, all nodes execute Algorithm 2 or 3 to find out whether they are in a cycle or not. After that, a cycle node gets assigned its current top choice and broadcasts a *remove* message which contains the assigned house to all nodes. When node P_i receives *remove* messages from other nodes, it deletes the houses contained in the messages from the preference list, i.e., from $pref_i$. When executing Algorithm 2 or 3, nodes might terminate at different rounds. Thus, we need to coordinate the execution of each stage. In order to achieve this, we use a convergecast step using the tree built in the execution of Algorithm 2 or 3. When a node completes broadcasting its *remove* message to all, it sends a *ok* message to its parent in the tree if it is a leaf node in the tree. For non-leaf nodes, they send an *ok* message to their parents only when they receive *ok* messages from all children. For the root node, when it receives *ok* from all its children, which means all nodes have updated their preference list, it broadcasts a *nextStage* message to all to notify all nodes to start the next stage of the algorithm.

Since each stage of Algorithm 4 simulates each iteration of the TTC mechanism. The correctness of Algorithm 4 follows from the correctness of TTC. We now look at the round and message complexity.

▶ **Theorem 21.** *Algorithm 4 computes the core of a housing market in $O(n)$ rounds and takes $O(n^2)$ messages.*

Proof. We just analyze the complexity of adopting the *Las Vegas* algorithm as a subroutine. The complexity of the deterministic algorithm just has an additional $\log^* n$ factor. Let l_i denote the length of the cycle at stage i of Algorithm 4. At stage i, both finding the cycle and convergecast along the tree need $O(\log l_i)$ rounds, w.h.p. Finding the cycle takes $O(n \log l_i)$ messages and convergecast takes $O(l_i)$ messages. Thus, each stage takes $O(\log l_i)$ randomized rounds and $O(n \log l_i)$ messages. Therefore, since $\sum l_i = n$, Algorith m 4 takes $O(n)$ rounds and $O(n^2)$ messages in the worst case. ◀

5 Conclusion

We conclude with two open problems. We have shown that computing a Pareto optimal matching for a housing allocation is in **CC**, which yields a linear time and linear work parallel algorithm. Computing an individually Pareto optimal matching for a housing market seems harder. It is interesting to know the relationship between this problem and the **CC** class. It is unlikely to be **CC**-complete, since this would imply a $\mathbf{RNC}^2$ and a quasi-$\mathbf{NC}^2$ algorithm for the **CC** class. We also show that computing the core of a housing market is **CC**-hard by giving a logspace reduction from the LFMM problem. It is interesting to know whether this problem is **CC**-complete. Or can we show that it is **P**-complete?

References

1 Atila Abdulkadiroğlu and Tayfun Sönmez. Random serial dictatorship and the core from random endowments in house allocation problems. *Econometrica*, 66(3):689–701, 1998.

2 Atila Abdulkadiroğlu and Tayfun Sönmez. House allocation with existing tenants. *Journal of Economic Theory*, 88(2):233–260, 1999.

3 David J Abraham, Katarína Cechlárová, David F Manlove, and Kurt Mehlhorn. Pareto optimality in house allocation problems. In *International Symposium on Algorithms and Computation*, pages 3–15. Springer, 2004.

4 Nir Amira, Ran Giladi, and Zvi Lotker. Distributed weighted stable marriage problem. In *International Colloquium on Structural Information and Communication Complexity*, pages 29–40. Springer, 2010.

5 Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32–53, 1986.

6 Stephen A Cook, Yuval Filmus, and Dai Tri Man Le. The complexity of the comparator circuit value problem. *ACM Transactions on Computation Theory (TOCT)*, 6(4):15, 2014.

7 Manlove David. *Algorithmics of matching under preferences*, volume 2. World Scientific, 2013.

8 Andrew Drucker, Fabian Kuhn, and Rotem Oshman. On the power of the congested clique model. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, pages 367–376. ACM, 2014.

9 Stephen Fenner, Rohit Gurjar, and Thomas Thierauf. Bipartite perfect matching is in quasi-NC. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pages 754–763. ACM, 2016.

10 Aanund Hylland and Richard Zeckhauser. The efficient allocation of individuals to positions. *Journal of Political economy*, 87(2):293–314, 1979.

11 Richard M Karp and Vijaya Ramachandran. Parallel algorithms for shared-memory machines, Handbook of Theoretical Computer Science (J. van Leeuwen, ed.), 1990.

12 Jinpeng Ma. Strategy-proofness and the strict core in a market with indivisibilities. *International Journal of Game Theory*, 23(1):75–83, 1994.

13 Bruce M Maggs and Ramesh K Sitaraman. Algorithmic nuggets in content delivery. *ACM SIGCOMM Computer Communication Review*, 45(3):52–66, 2015.

14 Ernst W Mayr and Ashok Subramanian. The complexity of circuit value and network stability. *Journal of Computer and System Sciences*, 44(2):302–323, 1992.

15 Gary L Miller and John H Reif. Parallel Tree Contraction and Its Application. Technical report, Harvard Univ Cambridge MA AIKEN Computation Lab, 1985.

16 Ketan Mulmuley, Umesh V Vazirani, and Vijay V Vazirani. Matching is as easy as matrix inversion. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 345–354. ACM, 1987.

17 Alvin E Roth. Incentive compatibility in a market with indivisible goods. *Economics letters*, 9(2):127–132, 1982.

18 Alvin E Roth and Andrew Postlewaite. Weak versus strong domination in a market with indivisible goods. *Journal of Mathematical Economics*, 4(2):131–137, 1977.

19 Lloyd Shapley and Herbert Scarf. On cores and indivisibility. *Journal of mathematical economics*, 1(1):23–37, 1974.

20 Uzi Vishkin. Randomized speed-ups in parallel computation. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 230–239. ACM, 1984.

21 Yufei Yuan. Residence exchange wanted: a stable residence exchange problem. *European Journal of Operational Research*, 90(3):536–546, 1996.

22 Xiong Zheng and Vijay Garg. Parallel and Distributed Algorithms for the housing allocation Problem. *arXiv preprint arXiv:1905.03111*, 2019.

23 Lin Zhou. On a conjecture by Gale about one-sided matching problems. *Journal of Economic Theory*, 52(1):123–135, 1990.

Oblivious Permutations on the Plane

Shantanu Das
Aix-Marseille University and CNRS, LIS, Marseille, France
shantanu.das@lis-lab.fr

Giuseppe A. Di Luna[1]
DIAG, University of Rome "Sapienza", Italy
diluna@diag.uniroma1.it

Paola Flocchini
University of Ottawa, Ottawa, Canada
paola.flocchini@uottawa.ca

Nicola Santoro
Carleton University, Ottawa, Canada
santoro@scs.carleton.ca

Giovanni Viglietta
JAIST, Nomi City, Japan
johnny@jaist.ac.jp

Masafumi Yamashita
Kyushu University, Fukuoka, Japan
masafumi.ymashita@gmail.com

Abstract

We consider a distributed system of n identical mobile robots operating in the two dimensional Euclidian plane. As in the previous studies, we consider the robots to be anonymous, oblivious, dis-oriented, and without any communication capabilities, operating based on the Look-Compute-Move model where the next location of a robot depends only on its view of the current configuration. Even in this seemingly weak model, most formation problems which require constructing specific configurations, can be solved quite easily when the robots are fully synchronized with each other. In this paper we introduce and study a new class of problems which, unlike the studied formation problems, cannot always be solved even in the fully synchronous model with atomic and rigid moves. This class of problems requires the robots to permute their locations in the plane. In particular, we are interested in implementing two special types of permutations – permutations without any fixed points and permutations of order n. The former (called Move-All) requires each robot to visit at least two of the initial locations, while the latter (called Visit-All) requires every robot to visit each of the initial locations in a periodic manner. We provide a characterization of the solvability of these problems, showing the main challenges in solving this class of problems for mobile robots. We also provide algorithms for the feasible cases, in particular distinguishing between one-step algorithms (where each configuration must be a permutation of the original configuration) and multi-step algorithms (which allow intermediate configurations). These results open a new research direction in mobile distributed robotics which has not been investigated before.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Computer systems organization → Robotics

Keywords and phrases Distributed Algorithms, Mobile Robots, Fully synchronous, Oblivious, Permutations, Chirality, Sequence of configurations

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.24

Related Version `https://arxiv.org/abs/1911.05239`

Funding *Paola Flocchini*: Natural Sciences and Engineering Research Council of Canada (N.S.E.R.C.) under the Discovery Grant program.

[1] Contact Author. Address: DIAG, Via Ariosto 25, Roma, Italy.

© Shantanu Das, Giuseppe A. Di Luna, Paola Flocchini, Nicola Santoro, Giovanni Viglietta, and Masafumi Yamashita;
licensed under Creative Commons License CC-BY
23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller; 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

The investigation of the computational and complexity issues arising in distributed systems of autonomous mobile robots is an important research topic in distributed computing. This has several applications, teams of robots could be sent to regions inaccessible to humans to perform a variety of tasks such as exploration and data-collection, monitoring, sensing or patrolling. Once deployed, the team of robots must coordinate with each other and perform the tasks autonomously without human intervention; this has motivated the design of distributed algorithms for coordination among the robots to enable them to perform the required tasks.

As a theoretical abstraction, the robots are usually viewed as computational entities modelled as points in a metric space, typically $\mathbb{R}^2$, in which they can move. The robots, identical and outwardly indistinguishable, have the same capabilities and execute the same (deterministic) algorithm. They can see each other, but cannot explicitly communicate with one another. This lack of direct communication capabilities means that the only means of interaction between robots are observations and movements: that is, communication is stigmergic. Each robot operates in "Look-Compute-Move" (LCM) cycles: during a cycle, it observes its surroundings, computes a destination point, and moves to it. Typically, the robots are assumed to have constant-size persistent memory or, more commonly, to be *oblivious* having no persistent memory: This paper assumes the latter model where robots in each cycle act only based on the current observation and have no memory of their activities from previous cycles. Further the robots do not have any means of orienting themselves; Each robot observes the location of other robots relative to its own position in the plane and the robots do not share any common coordinate system. If the robots agree on a common notion of clockwise direction, then we say the system has *chirality*.

Some typical problems that have been studied in this model include: *gathering* of robots (e.g., [11, 12]), uniform *dispersal*, *filling* a region with robots, *flocking*, etc. (for a review, see [6]). A generalization of some of these problems is that of *pattern formation*, where the n robots need to move from any initial configuration to a predefined pattern of n points in the plane. This class has been extensively studied (e.g., [1, 2, 5, 7, 8, 14, 15, 16, 17]). A major issue in such formation problems is the amount of symmetry (quantified by the notion of symmetricity [15]) in the starting configuration of robots and in the points of the pattern. In the arbitrary pattern formation problem, the points where the pattern is formed are *relative*, i.e. subject to rotation, translation and scaling of the input pattern. A different line of research is when the points of the pattern are *fixed*, a setting called *embedded pattern* and studied in [3, 9].

In some applications, forming a pattern may be the first step of a more complex task requiring coordination between robots. Consider, for example, robots that contain instrumentation for monitoring a site once there, as well as sensors for measurement (e.g., detecting traces of oil or precious metals, radioactivity, etc). If each robot has different sensors, the same site might need to be visited by all robots, and this must be done while still keeping all the sites monitored. A more relaxed version of this task is where each site must be visited by (at least) two robots. This task may be useful even in situations where all the robots contain the same sensors, e.g., if there are faulty sensors and we want to replicate the measurements.

These tasks are instances of a new class of problems quite different from the formation problems as the robots need to rotate among the given points of interests, forming permutations of a given pattern of points. We assume that each robot is initially occupying a point of interest (thus marking that location) and the objective is to permute the robots

among these locations periodically. The question is which permutations can be implemented starting from which patterns. We show a big difference between these classes of permutation problems compared to the formation problems studied previously. In particular, we show that even in the *fully synchronous* ($\mathcal{FSYNC}$) model, some of the permutation problems are not solvable, even when starting from configurations that admit a leader. In contrast, any formation problem (including gathering) is easily solvable in $\mathcal{FSYNC}$ when the starting configuration admits a leader.

Note that the permutation problems considered in this paper are *perpetual* tasks requiring continuous visits to the sites by the robots. Unlike the multiple pattern formation problem where robots continuously move from a pattern to the next [5], here the robots perpetually move but only exchanging locations in the same pattern. In particular, we focus on two interesting types of permutations — permutations without fixed points, and permutations of order n (i.e. n-cycles). These give rise to two specific problems (i) Move-All: every site must be visited by at least two robots and every robot has to visit at least two points, and, (ii) Visit-All: every robot must visit each of the points of interest. We provide a characterization of the solvability of these problems showing which patterns make it feasible to solve these problems and under what conditions. To the best of our knowledge, this is the first investigation on these class of problems.

Our Contributions. We distinguish between 1-step and multi-step algorithms; In the former case, we must form the permutations without passing through intermediate configurations, while in the latter case, a fixed number of intermediate configurations are allowed (see definitions in Section 2). We study 1-step and 2-step algorithms for Visit-All and Move-All, distinguishing the case when the robots share a common chirality from the case when they do not. We identify a special class of configurations denoted by $\mathcal{C}_\odot$, that are rotationally symmetric with exactly one robot in the center of symmetry. Such configurations do not always allow permutations without fixed points, thus making it difficult to solve the above problems.

We show that when there is chirality, the sets of initial configurations from which Visit-All and Move-All can be solved, using 1-step algorithms, are the same: that is, all configurations except those in $\mathcal{C}_\odot$ (Section 3). We then show that the characterization remains the same when we consider 2-step algorithms. Moreover, in the case of Visit-All, the solvability does not change even for k-step algorithms for any constant k.

On the other hand, when there is no chirality, we observe a difference between the solvability of Visit-All and Move-All. Configurations in $\mathcal{C}_\odot$ are clearly still non feasible for both problems. However, for the Move-All problem the class of unsolvable configurations also includes the ones where there exists a symmetry axis with a unique robot on it. On the other hand, the set of initial configurations from which Visit-All is solvable is different: the problem can be solved if and only if in the initial configuration there are no axes of symmetry or if there is a unique symmetry axis that does not contain any robots. Interestingly, also in this case, allowing 2-step algorithms does not change the set of solvable instances.

We then show that, when there is chirality and the coordinate systems of robots are visible (that is, a robot can sense the local coordinate system of the others), then Visit-All (and thus Move-All) is solvable from arbitrary initial configurations, and we provide a universal algorithm for solving the problems. Finally, we show that allowing a single bit of persistent memory per robot and assuming chirality, it is possible to solve the problems for all initial configurations (Section 6).

Due to the space constraint, some of the proofs and formal description of some algorithms have been omitted, they can be found in the full version.

2 Model, Definitions and Preliminaries

Robots and scheduler. We consider a set of dimensionless computational entities: the *robots*. These robots are modelled as points in the metric space $\mathbb{R}^2$; they are able to sense the environment detecting the presence of other robots, they can perform computations, and are able to move to any other point in the space. Each robot has its own local coordinate system centred in its own position (which may differ in orientation and unit distance from the coordinate system of other robots). For simplicity of description, we will use a global coordinate system S for analyzing the moves of the robots (robots themselves are unaware of this global system). Robots are *oblivious*: they do not have any persistent memory and thus, they cannot recall any information from previous computations. We indicate the set of robots with $R : \{r_0, r_1, \ldots, r_{n-1}\}$, however the robots themselves are not aware of the numbering assigned to them. All robots are identical and follow the same algorithm. We assume the so-called *Fully-Synchronous Scheduler* ($\mathcal{FSYNC}$). Under this scheduler, time can be seen as divided in discrete fixed length slots called *rounds*. In each round, each robot synchronously performs a *Look-Compute-Move* cycle [6]. During the *Look* phase, a robot r takes an instantaneous *snapshot* of the environment, the snapshot is an entire map of the plane containing positions of all the other robots with respect to the local coordinate system of r. During the *Compute* phase, robot r performs some local computation to decide its new destination point as a function of the aforementioned snapshot as input. Finally, in the *Move* phase, the robot moves to the computed destination point (which may be the same as current location).

Chirality. Robots may or may not share the same *handedness*: in the former case, they all agree on the clockwise direction and we say the system has *chirality* [6], in the latter case, robots do not have such an agreement and we say there is *no chirality*.

Configurations. A configuration C is an ordered tuple of points $C = (p_0, p_1, \ldots, p_{n-1})$, where $p_i = C[i]$ is the position of robot r_i in terms of the global coordinate system S. We denote by $Z = (Z_0, Z_1, \ldots Z_{n-1})$ the ordered tuple of coordinate systems where Z_i is the system used by robot r_i. Given a robot r_i located at p_i, we denote with $C \setminus \{r_i\}$ (or sometimes $C \setminus \{p_i\}$), the configuration obtained by removing robot r_i from C. We indicate with C_0 the initial configuration in which the robots start. We denote by $SEC(C)$ the smallest circle that encloses all points in the configuration C.

Symmetricity. Given any configuration C with robots having coordinate systems Z, the *symmetricity* $\sigma(C, Z) = m$ is the largest integer m such that the robots can be partitioned into classes of size at most m where robots in the same class have the same view (snapshot) in C (See [15, 16]). Alternatively, we can define the symmetricity (irrespective of Z) of a configuration as $\rho(C) = m$ where m is largest integer such that $\exists Z : \sigma(C, Z) = m$. For any configuration C, we have $\rho(C) \geq 1$, the configurations with $\rho(C) = 1$ are considered to be asymmetric (these are the only configurations that allow to elect a leader among robots). For symmetric configurations with $\rho(C) > 1$, C may have rotational symmetry with respect to the center c of $SEC(C)$, which coincides with the centroid of C in this case, or C may have mirror symmetry with respect to a line, called the axis of symmetry.

We define a special class of configurations denoted by $\mathcal{C}_\odot$. A configuration C is in $\mathcal{C}_\odot$, if and only if $\rho(C) = 1$, and there exists a unique robot r_c (the *central robot*) located at the center of $SEC(C)$ such that $\rho(C \setminus \{r_c\}) = k > 1$; In other words, C has a rotational

symmetry around r_c such that C can be rotated around centre r_c by an angle $\theta = \frac{\pi}{k}$ to obtain a permutation of C. Figure 2 is an example of a configuration in $\mathcal{C}_{\odot}$).

Permutations and runs. For a permutation $\pi = (\pi(0), \pi(1), \ldots, \pi(n-1))$ of $(0, 1, \ldots, n-1)$, define $\pi(C) = (p_{\pi(0)}, p_{\pi(1)}, \ldots, p_{\pi(n-1)})$. We denote: (1) the set of permutations with no fixed points as $\Pi_2 = \{\pi : \pi(i) \neq i,\ \forall i : 0 \leq i \leq n-1\}$ and (2) the set of cyclic permutations of order n as $\Pi_n = \{\pi : \pi^j(i) = i <=> nk = j \text{ for some } k \in \mathbb{N}\}$ where π^j indicates that we apply permutation π j times. Let $\Pi(C)$ be the set of all permutations of C.

Given an algorithm $\mathcal{A}$ and an initial configuration C_0 we denote any execution of algorithm A, starting with configuration C_0 as the run $\mathcal{R}_{\mathcal{A},C_0} : (C_0, C_1, C_2, \ldots)$, the infinite ordered sequence of configurations, where C_j is produced during round j of the execution.

Problem Definitions

We will study the following two problems:

- Move-All: An algorithm $\mathcal{A}$ is a 1-step solution algorithm for the Move-All problem, if every possible run of the algorithm $\mathcal{R}_{\mathcal{A},C_0} : (C_0, C_1, C_2, \ldots)$ is such that: $C_i = \pi^i(C_0)$ for some $\pi \in \Pi_2$. Intuitively, every configuration is a permutation of C_0 and in any two consecutive configurations, the position of each robot is different. As an extension for any $k \in \mathbb{N}^+$, a k-step solution requires that $C_{i \cdot k} = \pi^i(C_0)$ where $\pi \in \Pi_2$. (There is no constraint on the intermediate configurations C_j where k does not divide j.)
- Visit-All: An algorithm $\mathcal{A}$ is a 1-step solution algorithm for the Visit-All problem, if every possible run of the algorithm $\mathcal{R}_{\mathcal{A},C_0} : (C_0, C_1, C_2, \ldots)$ is such that: $C_i = \pi^i(C_0)$ for some $\pi \in \Pi_n$. Intuitively, every configuration is a permutation of C_0 and in every n consecutive configurations, every robot visit every location $p_i \in C_0$. We can similarly define a k-step solution for the problem where $C_{i \cdot k} = \pi^i(C_0)$ for some $\pi \in \Pi_n$.

Since $\Pi_n \subset \Pi_2$, it follows that any solution for Visit-All is also a solution to the Move-All problem.

Oblivious Permutations

Note that k-step solutions of Move-All and Visit-All specify that we must have a permutation of the initial configuration C_0 every k rounds. However, no constraint is given on the other *intermediate* configurations. Interestingly, when robots are oblivious the previous definitions imply a stronger version of the problem in which each configuration C_{j+k} has to be the permutation of configuration C_j that appeared k rounds ago.

▶ **Lemma 1.** *Let $\mathcal{A}$ be a k-step algorithm solving* Move-All *(or* Visit-All*), and let $\mathcal{R}_{\mathcal{A},C_0} : (C_0, C_1, C_2, \ldots)$ be any run of $\mathcal{A}$ starting from C_0. For each $j \in \mathbb{N}$ we have that $C_{j+k} = \pi(C_j)$ for some $\pi \in \Pi(C_j)$.*

Proof. We prove the lemma for Move-All, the extension to Visit-All is analogous and immediate. If $j = t \cdot k$ for some $t \in \mathbb{N}$ then the lemma follows from the problem definition. Thus let us consider a configuration C_j such that $j \neq t \cdot k$ for all $t \in \mathbb{N}$. We observe that $\mathcal{R}_{\mathcal{A},C_j}$ (that is a run of $\mathcal{A}$ starting from C_j) is equal to the suffix of $\mathcal{R}_{\mathcal{A},C_0}$ starting from C_j. This is due to the obliviousness of the robot, the fact that the algorithm is deterministic and the synchronous scheduler: starting from a certain configuration and an assignment of local coordinate systems, the algorithm will generate a fixed sequence of configurations. However in $\mathcal{R}_{\mathcal{A},C_j}$ we must have that $C_{j+k} = \pi(C_j)$ for some $\pi \in \Pi_2(C_j)$, otherwise $\mathcal{A}$ is not a correct algorithm for Move-All. ◀

3 Oblivious Robots with Chirality

In this section we consider robots having chirality (i.e., they agree on the same clockwise orientation).

3.1 1-Step Algorithms

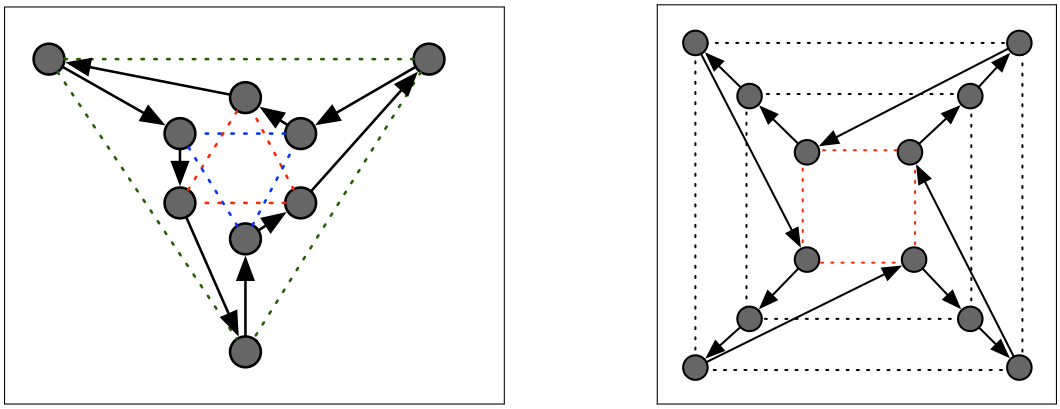
We first consider 1-step algorithms, and show that Move-All and Visit-All are solvable if the initial configuration C_0 is not in $\mathcal{C}_\odot$.

Intuition behind the solution algorithms. The underlining idea of our solution algorithms is to first make robots agree on a cyclic ordering of the robots, and then permute their positions according to this ordering. This algorithm is shown in Algorithm 1. When the center c of $SEC(C_0)$ is not occupied by any robot, we compute a cyclic ordering on the robots by taking the half-line passing through c and one of the robots closest to c and rotating it w.r.t. point c; the robots are listed in the order the line hits them. We can show that the ordering computed by any robot is a rotation of that computed by another robot (See Figure 1 for example). The only issue is when there is a robot at c. In this case, the robots compute a unique total order on the robots; this is always possible since $C_0 \notin \mathcal{C}_\odot$, which implies that C_0 is asymmetric and admits a total ordering.

From the above observations, it is immediate that the algorithm solves Visit-All: take a robot r, w.l.o.g. in position p_i, during n activations, the robot moves through all the robot positions in the computed cyclic order, returning back to p_i; thus, it has visited every point in C_0.

Algorithm 1 Visit-All Algorithm using a cyclic order.

1: Compute a cyclic order $(p_0, p_1, \ldots, p_{n-1})$ on C using Order(C).
2: If my position is p_i, set **destination** as $p_{(i+1) \mod n}$.



(a) An example of cyclic order induced by the Order algorithm.

(b) Another example of cyclic order induced by the Order algorithm.

Figure 1 Order algorithm: Examples of cyclic order computed by Algorithm 1.

▶ **Theorem 2.** *In systems with chirality, there exists a 1-step algorithm that solves* Visit-All *from any initial configuration $C_0 \notin \mathcal{C}_\odot$.*

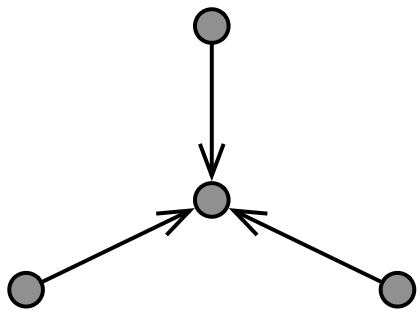


Figure 2 Configuration $C_0 \in \mathcal{C}_\odot$ where it is impossible to solve Move-All with a 1-step algorithm.

We now show that, when $C_0 \in \mathcal{C}_\odot$ Move-All (and thus Visit-All) is unsolvable by any 1-step algorithm.

▶ **Theorem 3.** *If $C_0 \in \mathcal{C}_\odot$ there exists no 1-step algorithm that solves* Move-All*, even when the robots have chirality.*

Proof. In any configuration in $\mathcal{C}_\odot$, the adversary can assign coordinate systems in such a way that each robot, except the central robot r_c, has at least one analogous with a symmetric view. This derives directly from the definition of $\mathcal{C}_\odot$. It is immediate to see that it is impossible to elect a unique robot to move to the center of C_0, taking the position of r_c. An example is given in Figure 2, where if one robot moves to the centroid of C_0, then every robot except r_c would do the same. This implies that, in the next round, it is impossible to form any $C_1 \in \Pi(C_0)$ with a different central robot. ◀

Note that Theorem 2 implies that Move-All is solvable under the same assumptions of the theorem (if we satisfy the Visit-All specification we satisfy also the Move-All specifications). Moreover, for the same reason, Theorem 3 implies that Visit-All is unsolvable. We can summarize the results of this section as follows:

▶ **Theorem 4.** *In systems with chirality,* Move-All *and* Visit-All *can be solved in 1-step if and only if $C_0 \notin \mathcal{C}_\odot$.*

3.2 2-step Algorithms

In light of Theorem 4, one may wonder what happens when multiple steps are allowed. In this section we show that allowing an intermediate step to reach the goal does not bring any advantages. We first introduce a technical lemma.

▶ **Lemma 5.** *Let $\mathcal{A}$ be a 2-step algorithm that solves* Move-All*. Starting from configuration $C_0 \in \mathcal{C}_\odot$, algorithm $\mathcal{A}$ cannot generate a run $\mathcal{R}_{\mathcal{A},C_0} : (C_0, C_1, C_2, C_3, \ldots)$ where $C_1 \in \mathcal{C}_\odot$.*

The above result is based on the observation that it is impossible to replace the central robot by another robot in 1-step. Thus the intermediate configuration must be a configuration $C_1 \notin \mathcal{C}_\odot$. Based on the above result, we can show the following:

▶ **Theorem 6.** *There exist no 2-step algorithm that solves* Move-All *from a configuration $C_0 \in \mathcal{C}_\odot$, even if the system has chirality.*

The informal idea here is that the central robot r_c in configuration C_0 needs to move away from the center to form the intermediate configuration C_1. However, in any 2-step algorithm, C_2 must be a permutation of C_0, with a different robot r' in the center. Now, following the same algorithm, robot r' would move away from the center to form the next configuration C_3. By choosing the coordinate systems of robots r_c and r' in an appropriate way, the adversary can ensure that C_3 would not be a permutation of C_1, thus violating the conditions in Lemma 1. This shows the impossibility.

Interestingly, when we consider Visit-All we can prove a stronger impossibility result that includes algorithms using any constant number of steps.

▶ **Theorem 7.** *There exists no k-step algorithm for* Visit-All*, starting from any configuration $C_0 \in \mathcal{C}_\odot$, where $k = \mathcal{O}(1)$. This result holds even if the system has chirality*

4 Oblivious Robots without Chirality

In this section we consider robots that do not share the same handedness. Interestingly, the absence of chirality changes the condition for solvability of Move-All and Visit-All, showing the difference between these two problems. This is due to the fact that in systems without chirality, the configuration of robots may have mirror symmetry, in addition to rotational symmetry as in the previous section.

4.1 Move-All

The following theorem illustrates the configurations for which the Move-All problem is unsolvable.

▶ **Theorem 8.** *In systems without chirality* Move-All *is unsolvable in* 1*-step starting from any configuration $C_0 \in \mathcal{C}_\odot$, as well as from any configuration that has a symmetry axis containing exactly one robot.*

We now consider the solutions to the Move-All problem for the feasible instances. If the configuration has a central symmetry (i.e., a rotational symmetry with $\theta = \pi$), each robot can be paired to its counterpart on the opposite end of the center, and the paired robots can swap positions. When the initial configuration has a rotational symmetry but no symmetry axes, then the robots can agree on a common chirality and the algorithms from the previous section can be applied. Thus the only remaining configurations are those with an axis of symmetry that are not central symmetric. For such configurations, it is possible to partition the robots in three disjoint subsets, and it make them move as follows: (see also Figure 3) (i) For the robots located on a symmetry axis, there exists a unique cyclic order on these robots. Robots on the axis are permuted according to this ordering. (ii) The second subset contains robots that are closer to one symmetry axis compared to other axes. These robots swap positions pairwise, each robot switching with its symmetric robot w.r.t. the closest axis. (iii) The last subset consists of robots that are equidistant from two distinct symmetry axes. Also in this case robots switch positions pairwise, and each one switches position with its symmetric robot w.r.t. the centroid c of configuration C_0.

For all the configurations excluded by Theorem 8, Move-All can be solved using the above approach.

▶ **Theorem 9.** *If $C_0 \notin \mathcal{C}_\odot$ and C_0 does not have a symmetry axis containing exactly one robot, then* Move-All *is solvable in* 1*-step even when the system does not have chirality.*

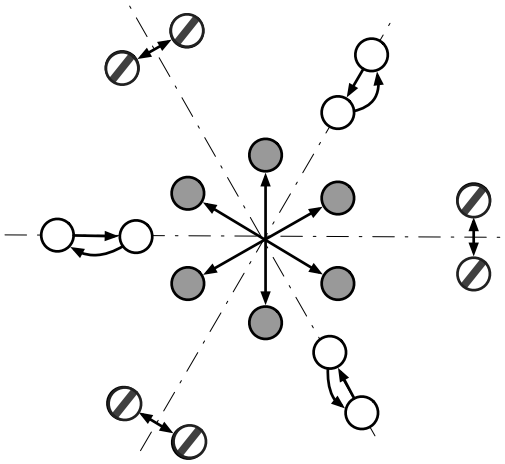


Figure 3 Configuration C_0 having 3 axes of symmetry. The arrows indicate three types of robot swaps: (1) Robots on the axis agree on a cyclic order (white robots); (2) Robots that are closer to one axis of symmetry swap positions w.r.t this axis (stripped robots); (3) Robots that are equidistant from two axes, swap position with symmetric robots w.r.t. the center (black robots).

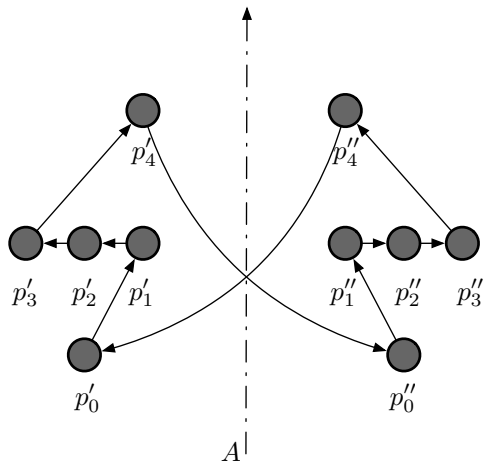


Figure 4 Configuration C_0 with a unique symmetry axis A and no robots on the axis. The arrow on axis A indicate the direction on which robots agree. The arrows among configuration points indicate the cyclic order induced by the algorithm.

To summarize, we have the following characterization for solvability of Move-All without chirality:

▶ **Theorem 10.** *In systems without chirality,* Move-All *is solvable in* 1*-step if and only if* $C_0 \notin \mathcal{C}_\odot$ *and* C_0 *does not have a symmetry axis containing exactly one robot.*

4.2 Visit-All

The Visit-All problem differs from Move-All only when $n > 2$, so we will assume in this section that $n \geq 3$. We will show that Visit-All is solvable without chirality if (i) $C_0 \notin \mathcal{C}_\odot$ and (ii) C_0 does not have symmetry axes, or there is a unique axis of symmetry that does not intersect any point of C_0. The main idea of the algorithm is the following. When C_0 does not have a symmetry axis, then it is possible to agree on a common notion of clockwise direction. Once this is done the algorithm from the previous section can be used. So we consider the case when C_0 has a unique axis of symmetry that does not intersect any point of C_0. We partition C_0 in two sets C' and C'', containing robots from the two sides of the axis of symmetry. In each of these sets it is possible to agree on a total order of the points (recall that the symmetry axis is unique). Let $[p'_0, p'_1, \ldots, p'_{\frac{n-1}{2}}]$ be the total order on C' and $[p''_0, p''_1, \ldots, p''_{\frac{n-1}{2}}]$ be the analogous order on C''. We obtain a cyclic order on C_0 by having element p''_0 following $p'_{\frac{n-1}{2}}$, and, in a symmetric way, p'_0 follows $p''_{\frac{n-1}{2}}$ (see Figure 4).

▶ **Theorem 11.** *When* $n > 2$ *and robots do not have chirality,* Visit-All *is solvable in* 1*-step if the initial configuration* $C_0 \notin \mathcal{C}_\odot$ *and either*

(i) *There are no symmetry axes in* C_0*, or,*

(ii) *There exists a unique symmetry axis of* C_0 *and no point of* C_0 *intersects the axis.*

Interestingly, without chirality, VISIT-ALL is not solvable if the assumptions of Th. 11 do not hold:

▶ **Theorem 12.** *When $n > 2$ and there is no chirality, there exists no algorithm that solves* VISIT-ALL *in 1-step from an initial configuration C_0 if one of the following holds:*
- $C_0 \in \mathcal{C}_\odot$
- *There exists a symmetry axis of C_0 intersecting a proper non-empty subset of C_0.*
- *There are at least two symmetry axes of C_0.*

To summarize, we have the following:

▶ **Theorem 13.** *In systems without chirality,* VISIT-ALL *is solvable in 1-step if and only if $C_0 \notin \mathcal{C}_\odot$ and either there are no symmetry axes in C_0, or there exists a unique symmetry axis that does not intersect any point of C_0.*

4.3 2-step Algorithms

We can show that 2-step algorithms do not help to enlarge the class of solvable configurations.

▶ **Theorem 14.** *When the system has no chirality,* MOVE-ALL *is not solvable in 2-steps, from an initial configuration C_0, if $C_0 \in \mathcal{C}_\odot$, or if there exists an axis of symmetry in C_0 containing a single robot.*

▶ **Theorem 15.** *When $n > 2$ and there is no chirality,* VISIT-ALL *is not solvable in 2-steps, from an initial configuration C_0, if one of the following holds:*
- $C_0 \in \mathcal{C}_\odot$
- *There exists a symmetry axis A of C_0 intersecting a proper non-empty subset of C_0.*
- *There are at least two symmetry axes of C_0.*

5 Oblivious Robots with Visible Coordinate Systems

In this section, we assume that each robot can see the coordinate system of all robots and the system has chirality. As we have seen in Section 3, with chirality, the only configurations in which VISIT-ALL cannot be solved are the ones in $\mathcal{C}_\odot$. We now present a VOTING algorithm that solves VISIT-ALL also starting from these configurations, provided that robots have this extra knowledge of the coordinate systems of other robots. The algorithm (see Algorithm 2) uses Procedure INNERPOLYGON, which takes a configuration C and returns only the points on the smallest non degenerate circle having the same center as $SEC(C)$ and passing through at least one point of C.

When $C_0 \notin \mathcal{C}_\odot$, the algorithm uses the ORDER procedure from Section 3. In case the initial configuration is in $\mathcal{C}_\odot$, the algorithm implements a voting procedure to elect a unique vertex of the innermost non-degenerate polygon P computed by Procedure INNERPOLYGON. The vote of a robot r is computed by translating its coordinate system to the center of $SEC(C_0)$. The vote of r will be given to the point of P that forms the smallest counter-clockwise angle with the x-axis of the translated system. Since the number of robots is co-prime to the size of P, a unique vertex (robot) can be elected and the elected point is used to break the symmetry and compute a total order among the robots. As before, the robots use this total order to move cyclically solving VISIT-ALL.

▶ **Theorem 16.** *If each robot can see the axes of the others and there is chirality, then there exists a 1-step algorithm solving* VISIT-ALL *for any initial configuration C_0.*

Algorithm 2 ORDER Algorithm when robots have visible coordinate systems.

procedure GETVOTE(Polygon P, robot r)
  $o = getCenter(P)$
  Consider the coordinate system with origin o and axes parallel to the system S_r of robot r. Let $p_v \in P$ be the point with smallest polar coordinates in this coordinate system.
  return p_v
procedure VOTING(Configuration C)
  $P = innerPolygon(C)$
  V=vector of size $|P|$ with all entries equal to 0.
  for all $r \in C$ **do**
    $r_v = getVote(P, r)$
    $V[v] = V[v] + 1$
  p_l = elect one robot in P using the votes in V.
  return p_l
procedure ORDER(Configuration C)
  if $C \in \mathcal{C}_\odot$ **then**
    $p_l = Voting(C)$
    Compute a cyclic order on positions in C using the leader robot p_l.
  else
    Compute an order using ORDER(C) of the algorithm in Section 3.1.
  return the cyclic order computed

6 Robots with one bit of Persistent Memory

Motivated by the impossibility result of Theorem 6, we now investigate non-oblivious robots having some persistent memory. Interestingly, we show that a single bit of memory is sufficient to overcome the impossibility, and solve VISIT-ALL using a 2-step algorithm. Note that we cannot overcome the impossibility using 1-step algorithms and Theorem 3, holds even if the robots are equipped with an infinite amount of memory.

We present the 2-step algorithm below (Algorithm 3) for $n \geq 3$ robots.

Algorithm 3 2-step VISIT-ALL with one bit of memory.

(Initially: $b = 0$)

if $C \notin \mathcal{C}_\odot \wedge b = 0$ **then**
  Compute an *order* using Algorithm from Section 3.
  Permute robots according to the computed *order*.
else if $C \in \mathcal{C}_\odot \wedge b = 0$ **then**
  b=1
  if I am the central robot **then**
    Compute a destination point **v** =COMPUTEMOVEMENTCENTRAL(C).
    set **destination** as **v**
else if $C \in \mathcal{C}_\odot \wedge b = 1$ **then**
  compute a destination point **v** =COMPUTEMOVEMENTNOTCENTRAL(C).
  set **destination** as **v**
else if $C \notin \mathcal{C}_\odot \wedge b = 1$ **then**
  $(C', p', Leader)$ =RECONSTRUCT(C)
  Compute a cyclic order $p_0, p_1, \ldots, p_{n-1}$ of positions in C' using the pivot robot p'.
  if I am the *Leader* **then**
    b=1
  else
    b=0
  if my position in C' was p_i **then**
    set **destination** as $p_{(i+1) \mod n}$

Intuitive description of the algorithm. The general idea is to use alternate rounds of communication and formation of the actual permutation. In the communication round, the robots create a special intermediate configuration that provides a total order on the robots;

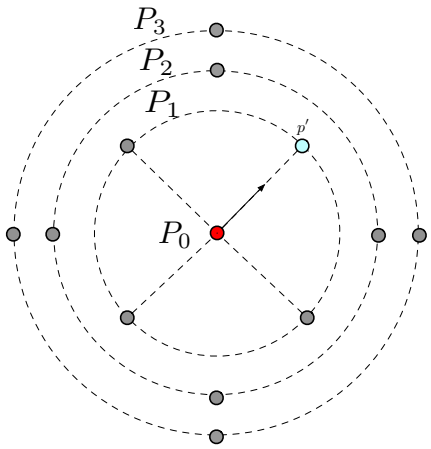


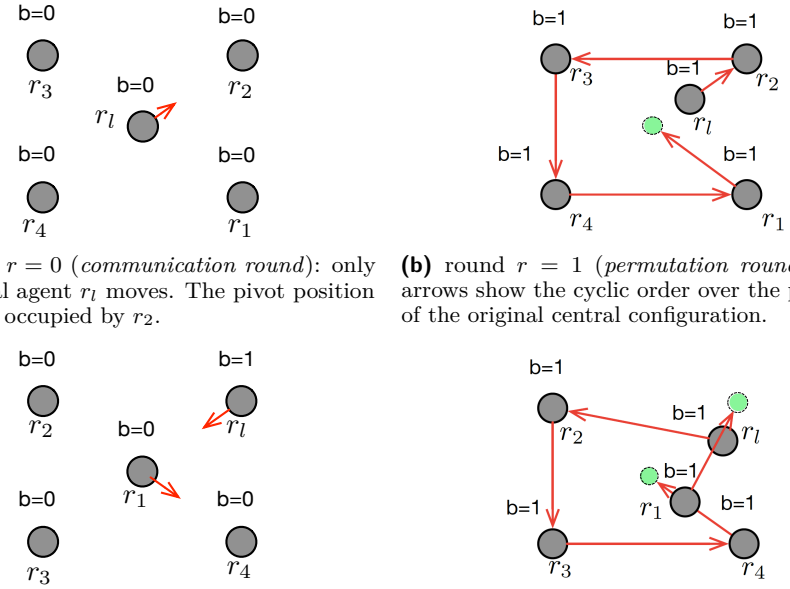
Figure 5 Case C2: the central robot $r_c = r_l$ moves to indicate the pivot point p' on circle P_1.

In the subsequent round they reconstruct the initial pattern forming the permutation of the initial configuration. The memory bit is crucial to distinguish the intermediate configuration from the initial configuration. If the initial configuration $C_0 \notin \mathcal{C}_\odot$ then the robots follow the 1-step algorithm described in Section 3 and we will show that this does not conflict with the rest of the algorithm designed for the case when $C_0 \in \mathcal{C}_\odot$, as described below.

Initially every robot has the bit b set to 0. When a robot observes that the configuration is in $\mathcal{C}_\odot$ and bit b is 0, it sets the bit to 1 to remember that the initial configuration $C_0 \in \mathcal{C}_\odot$. The central robot r_l in C_0 takes the role of *Leader* and performs a special move to create the intermediate configuration C_1 that is not in $\mathcal{C}_\odot$ but from C_1, it is possible to reconstruct the initial configuration C_0 or any permutation of it (This move is determined by procedure ComputeMovementCentral described in the next paragraph).

A key point of the algorithm is that the *Leader* robot remains invariant. At the next activation, the robots observe a configuration that is not in $\mathcal{C}_\odot$ and they have bit $b = 1$; this indicates that this is an intermediate configuration and the robots move to reconstruct a configuration $C_2 = \Pi(C_0)$. With the exception of the Leader r_l whose memory bit b is always set to 1, all the other robots will now reset their bit b to 0.

In the next round, the robots are in configuration C_2, where the central robot r_c is not r_l (the robots have performed one cyclic permutation). At this point, the robot r_l is the unique robot whose bit $b = 1$. All other robots have $b = 1$ and they behave similarly as in the first round, including robot r_c which moves like the central robot moved in C_0. However, the leader robot r_l also moves at the same time, in a special way (as described in procedure ComputeMovementNotCentral presented in the next paragraph). The combination of moves of the leader robot and the central robot allows the robots not only to determine the initial configuration, but also to uniquely identify a "pivot" point in the pattern (see Figure 6), which is kept invariant during the algorithm. The recognition of the pivot point allows the robots agree on the same cyclic ordering of the points in the initial pattern, thus allowing cyclic permutations of the robots.
An example execution of this algorithm is presented in Figure 6.



(a) round $r = 0$ (*communication round*): only the central agent r_l moves. The pivot position is the one occupied by r_2.

(b) round $r = 1$ (*permutation round*). The arrows show the cyclic order over the positions of the original central configuration.

(c) round $r = 2$ (*communication round*). r_l moves since it has $b = 1$ and r_1 moves because it is a central robot.

(d) round $r = 3$ (*permutation round*). The robots move according to the cyclic order on the reconstructed central configuration.

Figure 6 Example execution of the first 4 rounds for the case $n > 3$.

Movements of r_l and r_c. We now describe details of procedures ComputeMovementCentral and ComputeMovementNotCentral used in Algorithm 3. The movements of the robot leader r_l and the central robot r_c (if different from r_l) have to be designed in such a way as to break the symmetry of the configuration by electing always the same pivot position p', and to make it possible to reconstruct the original configuration after the move. For any configuration $C \in \Pi(C_0)$. we define $P_0, P_1, \ldots, P_m$ as a decomposition of C into concentric circles, where P_0 is the degenerate circle consisting of only the central robot, P_1 is the innermost non-degenerate circle on which p' is located, and finally $P_m = SEC(C)$ (see Figure 5).

Procedure ComputeMovementCentral determines the movement of robot r_c, according to the number of robots n.

- (Case C1: $n = 3$): In such a case, the robots are on a single line. Let s be the segment of this line containing all three robots. Robot r_c moves perpendicularly to s of a distance $d = \frac{|s|}{2}$. The direction of movement is chosen such that the pivot position p' will be the position of the first robot in the clockwise direction from r_c.
- (Case C2: $n > 3$): Robot r_c chooses a robot position p' on P_1 as the pivot point. Robot r_c moves on the segment connecting r_c and p' by a very small distance (much smaller compared to the radius of P_1). See Figure 5.

Procedure ComputeMovementNotCentral computes the movements of the leader robot r_l, when different from r_c, again according to the value of n:

- (Case L1: $n = 3$): Note that robots are on a single line. Robot r_l moves along s by a small distance, changing segment s to s'. The pivot position p' will be indicated by the direction that goes from the center of s to the new center of s'.
- (Case L2: $n > 3$): We have three sub-cases depending on the circle P_j which contains robot r_l, and on the number of other robots on P_j. Recall that $r_l \notin P_0$. We treat each P_j

as a set, e.g. $|P_j|$ indicates the number of robots in P_j. Let x be the difference between the radii of P_{j-1} and P_j, and let h be the segment connecting P_0 and robot r_l. Let p be the position of the first robot on P_1 encountered by walking in clockwise direction starting from the point of intersection between h and P_1. Let *nhop* be the number of robots in P_1 between p and the pivot point p'. Recall that $|P_m| > 1$, since $P_m = SEC(C)$ and C is rotationally symmetric. In the following, *encode* is an appropriately chosen function from $\mathbb{N}$ to $(\frac{1}{2}, 1)$.

- (Sub-case L2.1): When $P_j \neq P_m$ or $P_j = P_m$ and $|P_m| > 3$: Robot r_l moves on h towards P_j by a quantity $encode(nhop) * \frac{x}{2}$.
- (Sub-case L2.2): When $P_j = P_m$ and $|P_m| = 3$: Note that P_m has to be rotationally symmetric; therefore it contains 3 robots each of them forming an angle of $\frac{2\pi}{3}$ with its adjacent neighbours. Robot r_l moves to a point of P_j that creates with its counter-clockwise neighbour an angle that is $encode(nhop) * \frac{2\pi}{3}$.
- (Sub-case L2.3): When $P_j = P_m$ and $|P_m| = 2$: let s be the segment connecting the two robots on P_j. Robot r_l moves on s, expanding P_j in such a way that the new diameter is $2D + encode(nhop) * D$.

It is easy to see that after r_c and r_l move according to the above procedures, the resulting intermediate configuration C' is not in $\mathcal{C}_\odot$. We now show how to reconstruct the initial configuration C from C'.

Reconstruction of the initial configuration. When the current configuration C' is not in $\mathcal{C}_\odot$ and the robots have bit $b = 1$, the robots know that they are in an intermediate configuration and they have to (1) reconstruct the original configuration C, (2) determine the pivot point p' in C, and (3) identify the leader robot r_l. The reconstruction is performed by procedure Reconstruct, which again, depends on the value of n.

- If $n = 3$: the robots must form a triangle. The base of the triangle is its largest edge e. The algorithm computes the ratio of the height of the triangle over the length of base e to determine if r_l was r_c or not.
  - If the height of the triangle is exactly half of e, the algorithm infers that $r_l = r_c$ and that the two other robots are the endpoints of e (case C1). The pivot point p' is the position of the first clockwise robot starting from the top of the triangle. The original configuration C is easily reconstructed: the endpoints of e are in the same position, and the central robot will be in the intersection of the perpendicular segment that goes through r_l and e.
  - If the height is slightly less, or slightly more, than the largest edge e, then the algorithm infers that r_l was one of the endpoint; we are in case (L1.1). The reconstruction of C is simple: take the intersection x of the perpendicular segment that goes through r_l and e, the position of the endpoints of C is reconstructed using the fact that the height of the triangle is exactly half of the original segment, and that x was the center of the original segment. Robot r_l is the endpoint that moved, and the pivot p' can be computed by evaluating if r_l moved towards or away from the old center.
- If $n > 3$: The algorithm starts by examining P_m, in order to understand if r_l was on the SEC and executed the sub-case (L2.2) or (L2.3). If the test is negative it proceeds using an "onion peeling" approach, in which the algorithm, starting from the outermost P_m, progressively examines each P_j until it finds an asymmetry or it reaches P_0. The onion peeling proceeds by first computing the SEC, that is P_m, and then computing each P_j by considering progressively smaller concentric circles.
  - Test for case (L2.3): This test case is done only on P_m. If the center of P_m is not contained in $\mathrm{SEC}(C' \setminus P_{m-1})$, then the algorithm detects case (L2.3). P_m is adjusted

to a new one that has the diameter equal to the distance between the two furthest robots on P_m. Robot r_l will be the robot on P_m that is farthest from robots in P_{m-1}. The reconstruction of the last layer is done by knowing that it will be a circle with the same center of $\mathrm{SEC}(C' \setminus P_{m-1})$ that passes through $P_m \setminus \{r_l\}$ and finally p' will be indicated by decoding the information encoded in the diameter of P_m.

- Test for case (L2.2): This test case is done only on P_m. If $|P_m| = 3$ and it is not rotationally symmetric, and ($|P_{m-1}| > 1$ or $m - 1 = 0$), then the algorithm detects case (L2.2). Robot r_l is one that is not forming an angle of $\frac{2\pi}{3}$ radians with any of its adjacent neighbours, position p' is encoded in the smallest angle that r_l is forming. The original position of r_l can be easily reconstructed: it is the one that forms an angle of $\frac{2\pi}{3}$ with each of its adjacent robots.
- Test for case (L2.1): This test case is done on layers different than P_m. If $|P_j| = 1$ then the algorithm detects case (L2.1): robot r_l is the only robot in P_j and it is trivial to reconstruct C and compute the pivot p'.

If the algorithm reaches P_0 without finding any asymmetry, then we have that $r_l = r_c$ (case C2). The decoding is trivial in this case: the original position of r_l is the center of P_1, and the pivot position p' is in the direction where r_l moved.

Based on the above discussion, we conclude with the following result:

▶ **Theorem 17.** *There exists an universal algorithm to solve* Visit-All *for robots with* 1 *bit of persistent memory when the system has chirality.*

7 Concluding Remarks

To the best of our knowledge, this is the first investigation of the problems of permuting the positions of a set of mobile robots in the plane. Surprisingly this class of problems seems to be more difficult than the previously studied problems such as gathering and pattern formation, which have easy solutions for the strongest model of fully synchronous robots with rigid movements. Thus the characterization of solvable instances for permutation problems is quite different as shown in this paper. Moreover we also showed that being non-oblivious is helpful for permuting robots, unlike the formation problems where the solvability is unaffected by obliviousness [16]. The paper opens several research directions that are worth investigating: an interesting direction would be to discover other class of problems which cannot be solved even when it is easy to elect a leader (as the class of problems considered here). The difficulty in solving the permutation problems seems to be unrelated to agreement problems such as leader election. In particular we may try to study the differences between leader election and permutation problems and determine if the latter is strictly more difficult than the former. We may also consider other interesting assumptions that can help in overcoming the challenges for permuting robots without orientation. One possibility is the investigation of robots with the additional capability of communicating using visible lights [4, 10, 13].

References

1 H Ando, I Suzuki, and M Yamashita. Formation and agreement problems for synchronous mobile robots with limited visibility. In *Proceedings of Tenth International Symposium on Intelligent Control*, pages 453,460. IEEE, 1995.

2 Quentin Bramas and Sébastien Tixeuil. Brief Announcement: Probabilistic Asynchronous Arbitrary Pattern Formation. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, pages 443–445, 2016. doi:10.1145/2933057.2933074.

3 Serafino Cicerone, Gabriele Di Stefano, and Alfredo Navarra. Asynchronous Embedded Pattern Formation Without Orientation. In *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings*, pages 85–98, 2016. doi:10.1007/978-3-662-53426-7_7.

4 Shantanu Das, Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Masafumi Yamashita. Autonomous mobile robots with lights. *Theor. Comput. Sci.*, 609:171–184, 2016. doi:10.1016/j.tcs.2015.09.018.

5 Shantanu Das, Paola Flocchini, Nicola Santoro, and Masafumi Yamashita. Forming sequences of geometric patterns with oblivious mobile robots. *Distributed Computing*, 28(2):131–145, 2015. doi:10.1007/s00446-014-0220-9.

6 Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro. *Distributed Computing by Oblivious Mobile Robots*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2012. doi:10.2200/S00440ED1V01Y201208DCT010.

7 Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Peter Widmayer. Arbitrary pattern formation by asynchronous, anonymous, oblivious robots. *Theor. Comput. Sci.*, 407(1-3):412–447, 2008. doi:10.1016/j.tcs.2008.07.026.

8 Nao Fujinaga, Yukiko Yamauchi, Shuji Kijima, and Masafumi Yamashita. Asynchronous Pattern Formation by Anonymous Oblivious Mobile Robots. In *Distributed Computing - 26th International Symposium, DISC 2012, Salvador, Brazil, October 16-18, 2012. Proceedings*, pages 312–325, 2012. doi:10.1007/978-3-642-33651-5_22.

9 Nao Fujinaga, Yukiko Yamauchi, Hirotaka Ono, Shuji Kijima, and Masafumi Yamashita. Pattern Formation by Oblivious Asynchronous Mobile Robots. *SIAM J. Comput.*, 44(3):740–785, 2015. doi:10.1137/140958682.

10 Giuseppe Antonio Di Luna, Paola Flocchini, Sruti Gan Chaudhuri, Federico Poloni, Nicola Santoro, and Giovanni Viglietta. Mutual visibility by luminous robots without collisions. *Inf. Comput.*, 254:392–418, 2017. doi:10.1016/j.ic.2016.09.005.

11 Giuseppe Antonio Di Luna, Paola Flocchini, Nicola Santoro, and Giovanni Viglietta. TuringMobile: A Turing Machine of Oblivious Mobile Robots with Limited Visibility and Its Applications. In *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*, pages 19:1–19:18, 2018. doi:10.4230/LIPIcs.DISC.2018.19.

12 Giuseppe Antonio Di Luna, Paola Flocchini, Nicola Santoro, Giovanni Viglietta, and Masafumi Yamashita. Meeting in a Polygon by Anonymous Oblivious Robots. In *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, pages 14:1–14:15, 2017. doi:10.4230/LIPIcs.DISC.2017.14.

13 Giuseppe Antonio Di Luna and Giovanni Viglietta. Robots with Lights. In *Distributed Computing by Mobile Entities, Current Research in Moving and Computing*, pages 252–277. Springer, 2019. doi:10.1007/978-3-030-11072-7_11.

14 Kazuo Sugihara and Ichiro Suzuki. Distributed algorithms for formation of geometric patterns with many mobile robots. *J. Field Robotics*, 13(3):127–139, 1996. doi:10.1002/(SICI)1097-4563(199603)13:3<127::AID-ROB1>3.0.CO;2-U.

15 Ichiro Suzuki and Masafumi Yamashita. Distributed Anonymous Mobile Robots: Formation of Geometric Patterns. *SIAM J. Comput.*, 28(4):1347–1363, 1999. doi:10.1137/S009753979628292X.

16 Masafumi Yamashita and Ichiro Suzuki. Characterizing geometric patterns formable by oblivious anonymous mobile robots. *Theor. Comput. Sci.*, 411(26-28):2433–2453, 2010. doi:10.1016/j.tcs.2010.01.037.

17 Yukiko Yamauchi and Masafumi Yamashita. Randomized Pattern Formation Algorithm for Asynchronous Oblivious Mobile Robots. In *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings*, pages 137–151, 2014. doi:10.1007/978-3-662-45174-8_10.

On Memory, Communication, and Synchronous Schedulers When Moving and Computing

Paola Flocchini
EECS, University of Ottawa, Canada
pflocchi@uottawa.ca

Nicola Santoro
School of Computer Science, Carleton University, Canada
santoro@scs.carleton.ca

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

Abstract

We investigate the computational power of distributed systems whose autonomous computational entities, called robots, move and operate in the 2-dimensional Euclidean plane in synchronous *Look-Compute-Move* (*LCM*) cycles. Specifically, we focus on the power of persistent memory and that of explicit communication, and on their computational relationship.

In the most common model, $\mathcal{OBLOT}$, the robots are oblivious (no persistent memory) and silent (no explicit means of communication). In contrast, in the $\mathcal{LUMI}$ model, each robot is equipped with a constant-sized persistent memory (called *light*), visible to all the robots; hence, these luminous robots are capable in each cycle of both remembering and communicating. Since luminous robots are computationally more powerful than the standard oblivious one, immediate important questions are about the individual computational power of persistent memory and of explicit communication. In particular, which of the two capabilities, memory or communication, is more important? in other words, is it better to remember or to communicate ?

In this paper we address these questions, focusing on two sub-models of $\mathcal{LUMI}$: $\mathcal{FSTA}$, where the robots have a constant-size persistent memory but are silent; and $\mathcal{FCOM}$, where the robots can communicate a constant number of bits but are oblivious. We analyze the relationship among all these models and provide a complete exhaustive map of their computational relationship. Among other things, we prove that communication is more powerful than persistent memory under the fully synchronous scheduler FSYNCH, while they are incomparable under the semi-synchronous scheduler SSYNCH.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Look-Compute-Move, Oblivious mobile robots, Robots with lights, Memory versus Communication, Moving and Computing

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.25

Acknowledgements This research was partly supported by NSERC through the Discovery Grant program, by Prof. Flocchini's University Research Chair, by JSPS KAKENHI No. 17K00019, and by Japan Science and Technology Agency (JST) SICORP Grant#JPMJSC1806.

1 INTRODUCTION

1.1 Background and Motivation

The computational issues of autonomous mobile entities operating in an Euclidean space in *Look-Compute-Move* (*LCM*) cycles have been the object of much research in distributed computing. In the *Look* phase, an entity, viewed as a point and usually called *robot*, obtains a snapshot of the space; in the *Compute* phase it executes its algorithm (the same for all

© Paola Flocchini, Nicola Santoro, and Koichi Wada;
licensed under Creative Commons License CC-BY
23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller; Article No. 25; pp. 25:1–25:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

robots) using the snapshot as input; it then moves towards the computed destination in the *Move* phase. Repeating these cycles, the robots are able to collectively perform some tasks and solve some problems. The research interest has been on determining the impact that *internal* capabilities (e.g., memory, communication) and *external* conditions (e.g. synchrony, activation scheduler) have on the solvability of a problem.

In the most common model, $\mathcal{OBLOT}$, in addition to the standard assumptions of *anonymity* and *uniformity* (robots have no IDs and run identical algorithms), the robots are *oblivious* (no persistent memory to record information of previous cycles) and *silent* (without explicit means of communication). Computability in this model has been the object of intensive research since its introduction in [27]. Extensive investigations have been carried out to clarify the computational limitations and powers of these robots for basic coordination tasks such as Gathering (e.g., [1, 2, 4, 6, 7, 8, 15, 21, 27]), Pattern Formation (e.g., [16, 18, 27, 30, 31]), Flocking (e.g., [5, 19, 26]); for a recent account of the state of the art on some of these problems, see [13] and the chapters therein. Clearly, the restrictions created by the absence of persistent memory and the incapacity of explicit communication severely limits what the robots can do and renders complex and difficult for them to perform the tasks they can do.

A model where robots are provided with some (albeit limited) persistent memory and communication means is the $\mathcal{LUMI}$ model, formally defined and analyzed in [9, 10], following a suggestion in [24]. In this model, each robot is equipped with a constant-sized memory (called *light*), whose value (called *color*) can be set during the *Compute* phase. The light is visible to all the robots and is persistent in the sense that it is not automatically reset at the end of a cycle. Hence, these luminous robots are capable in each cycle of both remembering and communicating a constant number of bits. There is a lot of research work on the design of algorithms and the feasibility of problems for luminous robots (e.g., [3, 10, 11, 17, 20, 22, 23, 25, 28, 29]); for a recent survey, see [12].

As for the computational relationship between $\mathcal{OBLOT}$ and $\mathcal{LUMI}$, the availability of both persistent memory and communication, however limited, clearly renders luminous robots more powerful than oblivious robots (e.g., [10]). This immediately raises important questions about the individual computational power of the two internal capabilities: memory and communication. In particular,

- if the robots were endowed with a constant number of bits of persistent memory but were still unable to communicate explicitly, what problems could they solve ?
- If the robots could communicate a constant number of bits in each cycle, but were oblivious, what would be their computational power then ?
- Which of the two capabilities, memory or communication, is more important? or, in other words, *is it better to remember or to communicate ?*

Helpful in this regards are two sub-models of $\mathcal{LUMI}$. In the first model, $\mathcal{FSTA}$, the light of a robot is visible only by that robot, while in the second model, $\mathcal{FCOM}$, the light of a robot is visible only to the other robots. Thus in $\mathcal{FSTA}$ the color merely encodes an internal state; hence the robots are *finite-state* and *silent*. On the contrary, in $\mathcal{FCOM}$, a robot can communicate to the other robots through its colored light but forgets the content of its transmission by the next cycle; that is, robots are *finite-communication* and *oblivious*.

This means that some answers to the above questions, as well as others, can be provided by exploring and determining the computational power within these four models, $\mathcal{OBLOT}$, $\mathcal{FSTA}$, $\mathcal{FCOM}$, and $\mathcal{LUMI}$ and with respect to each other. This is the focus of this paper.

When studying computability within a model of LCM robots, two interrelated external factors play a crucial role: *time* and *activation schedule*. With respect to these factors, there are two fundamentally different settings: *asynchronous* and *synchronous*.

In the *asynchronous* setting (Asynch), first studied in [14], there is no common notion of time, each robot is activated independently of the others, the duration of each phase is finite but unpredictable and might be different in different cycles.

In the *synchronous* setting (Ssynch), also called semi-synchronous and first studied in [27], time is divided into discrete intervals, called *rounds*; in each round some (possibly all) robots are activated, perform their LCM cycle simultaneously, and terminate by the end of the round. The selection of which robots are activated at a round is made by the adversarial scheduler, constrained to be fair. A special synchronous setting which plays an important role is the *fully-synchronous* setting (Fsynch) where every robot is activated in every round; that is, the activation scheduler has no adversarial power.

Returning to the focus of this paper, which is to understand the computational power within each model, the amount of available knowledge is rather limited. In particular, it is known that, within $\mathcal{OBLOT}$, robots in Fsynch are strictly more powerful than those in Ssynch: there are problems solvable in Fsynch but unsolvable in Ssynch [27]. It is also known that, within $\mathcal{LUMI}$, robots have in Asynch the same computational power as in Ssynch [10]. As for the relationship between different models, it has been shown that asynchronous luminous robots are strictly more powerful than oblivious synchronous robots [10]. The $\mathcal{FCOM}$ and $\mathcal{FSTA}$ models have been studied only in the context of *Rendezvous*, which cannot be solved in Ssynch in the $\mathcal{OBLOT}$ model, while it has been shown to be solvable in both $\mathcal{FCOM}$ and $\mathcal{FSTA}$ [17]. In this paper we investigate these questions, focusing on synchronous schedulers.

1.2 Contributions

We analyze the relationship among all these models and provide a complete exhaustive map of their computational relationship, summarized in Tables 1-3, where: $\mathcal{X}^Y$ denotes model $\mathcal{X}$ under scheduler Y; F and S stand for Fsynch and Ssynch respectively, $A > B$ indicates that model A is computationally more powerful than model B, $A \equiv B$ denotes that they are computationally equivalent, $A \perp B$ denotes that they are computationally incomparable.

We first examine the computational relationship within each scheduler. Among other things, we prove that the answer to the question *"is it better to remember or to communicate ?"* depends on the type of scheduler. More precisely, communication is more powerful than persistent memory if the scheduler is fully synchronous; on the other hand, the two models are incomparable under the semi-synchronous scheduler.

We then focus on the relationship between Fsynch and Ssynch. In addition to the expected dominance results, we prove some interesting orthogonality results. In fact, we show that, on one hand, both $\mathcal{FSTA}^S$ and $\mathcal{FCOM}^S$ are incomparable with $\mathcal{OBLOT}^F$, on the other $\mathcal{LUMI}^S$ is incomparable with $\mathcal{FSTA}^F$, $\mathcal{FCOM}^F$, and even with $\mathcal{OBLOT}^F$. We also close an open problem of [10].

Table 1 Relationships within Fsynch.

	$\mathcal{FCOM}^F$	$\mathcal{FSTA}^F$	$\mathcal{OBLOT}^F$
$\mathcal{LUMI}^F$	$\equiv$ (Th.2)	$>$ (Th.2,6)	$>$ (Th.2,6,10)
$\mathcal{FCOM}^F$	–	$>$ (Th.6)	$>$ (Th.6,10)
$\mathcal{FSTA}^F$	–	–	$>$ (Th.10)

Table 2 Relationships within Ssynch.

	$\mathcal{FCOM}^S$	$\mathcal{FSTA}^S$	$\mathcal{OBLOT}^S$
$\mathcal{LUMI}^S$	$>$ (Th.17)	$>$ (Th.17)	$>$ (Th.15, 17)
$\mathcal{FCOM}^S$	–	$\perp$ (Th.14)	$>$ (Th.15)
$\mathcal{FSTA}^S$	–	–	$>$ (Th.15)

Table 3 Relationship between FSYNCH and SSYNCH.

	$\mathcal{LUMI}^S$	$\mathcal{FCOM}^S$	$\mathcal{FSTA}^S$	$\mathcal{OBLOT}^S$
$\mathcal{LUMI}^F$ $\equiv \mathcal{FCOM}^F$	$>$ (Th.20)	$>$ (Th.20)	$>$ (Th.6,20)	$>$ (Th.15,20)
$\mathcal{FSTA}^F$	$\perp$ (Th.26)	$\perp$ (Th.26)	$>$ (Th.20)	$>$ (Th.15,20)
$\mathcal{OBLOT}^F$	$\perp$ (Th.28)	$\perp$ (Th.25)	$\perp$ (Th.25)	$>$ (Th.20)

2 MODELS AND PRELIMINARIES

2.1 The Basics

The systems considered in this paper consist of a team $R = \{r_0, \cdots, r_{n-1}\}$ of computational entities moving and operating in the Euclidean plane $\mathbb{R}^2$. Viewed as points and called *robots*, the entities can move freely and continuously in the plane. Each robot has its own local coordinate system and it always perceives itself at its origin; there might not be consistency between these coordinate systems. A robot is equipped with sensorial devices that allows it to observe the positions of the other robots in its local coordinate system.

The robots are *identical*: they are indistinguishable by their appearance and they execute the same protocol. The robots are *autonomous*, without a central control.

At any point in time, a robot is either *active* or *inactive*. Upon becoming active, a robot r executes a *Look-Compute-Move* (*LCM*) cycle performing the following three operations:

1. *Look:* The robot activates its sensors to obtain a snapshot of the positions occupied by robots with respect to its own coordinate system[1].
2. *Compute:* The robot executes its algorithm using the snapshot as input. The result of the computation is a destination point.
3. *Move:* The robot moves to the computed destination[2]. If the destination is the current location, the robot stays still.

When inactive, a robot is idle. All robots are initially idle. The amount of time to complete a cycle is assumed to be finite, and the *Look* operation is assumed to be instantaneous.

Let $x_i(t)$ denote the location of robot r_i at time t in a global coordinate system (unknown to the robots), and let $X(t) = \{x_i(t) : 0 \leq i \leq n-1\} = \{x_0(t), x_1(t), \ldots, x_{m-1}(t)\}$; observe that $|X(t)| = m \leq n$ since several robots might be at the same location at time t.

In this paper, we do not assume that the robots have a common coordinate system. If they agree on the same circular orientation of the plane (i.e., they do agree on "clockwise" direction), we say that there is *chirality*. Except when explicitly stated, we assume there is chirality.

2.2 The Models

Different models, based on the same basic premises defined above, have been considered in the literature and will be examined here.

[1] This is called the *full visibility* (or unlimited visibility) setting; restricted forms of visibility have also been considered for these systems

[2] This is called the *rigid mobility* setting; restricted forms of mobility (e.g., when the movement may be interrupted by an adversary) have also been considered for these systems

In the most common model, $\mathcal{OBLOT}$, the robots are *silent*: they have no explicit means of communication; furthermore they are *oblivious*: at the start of a cycle, a robot has no memory of observations and computations performed in previous cycles.

In the other common model, $\mathcal{LUMI}$, each robot r is equipped with a persistent visible state variable $Light[r]$, called *light*, whose values are taken from a finite set C of states called *colors* (including the color that represents the initial state when the light is off). The colors of the lights can be set in each cycle by r at the end of its *Compute* operation. A light is *persistent* from one computational cycle to the next: the color is not automatically reset at the end of a cycle; the robot is otherwise oblivious, forgetting all other information from previous cycles. In $\mathcal{LUMI}$, the *Look* operation produces a colored snapshot; i.e., it returns the set of pairs $(position, color)$ of the other robots[3]. Note that if $|C| = 1$, then the light is not used; thus, this case corresponds to the $\mathcal{OBLOT}$ model.

It is sometimes convenient to describe a robot r as having $k \geq 1$ lights, denoted $r.light_1, \ldots, r.light_k$, where the values of $r.light_i$ are from a finite set of colors C_i, and to consider $Light[r]$ as a k-tuple of variables; clearly, this corresponds to r having a single light that uses $\Pi_{i=1}^{k}|C_i|$ colors.

The lights provide simultaneously persistent memory and direct means of communication, although both limited to a constant number of bits per cycle. Two sub-models of $\mathcal{LUMI}$ have been defined and investigated, each offering only one of these two capabilities.

In the first model, $\mathcal{FSTA}$, a robot can only see the color of its own light; that is, the light is an *internal* one and its color merely encodes an internal state. Hence the robots are *silent*, as in $\mathcal{OBLOT}$; but are *finite-state*. Observe that a snapshot in $\mathcal{FSTA}$ is the same as in $\mathcal{OBLOT}$.

In the second model, $\mathcal{FCOM}$, the lights are *external*: a robot can communicate to the other robots through its colored light but forgets the color of its own light by the next cycle; that is, robots are *finite-communication* but *oblivious*. A snapshot in $\mathcal{FCOM}$ is like in $\mathcal{LUMI}$ except that, for the position x where the robot r performing the *Look* is located, $Light[r]$ is omitted from the set of colors present at x.

In all the above models, a *configuration* $C(t)$ at time t is the multi-set of the n pairs of the $(x_i(t), c_i(t))$, where $c_i(t)$ is the color of robot r_i at time t.

2.3 The Schedulers

With respect to the activation schedule of the robots, and the duration of their *Look-Compute-Move* cycles, the fundamental distinction is between the *asynchronous* and *synchronous* settings.

In the *asynchronous* setting (Asynch), first studied in [14], there is no common notion of time, each robot is activated independently of the others, the duration of each phase is finite but unpredictable and might be different in different cycles.

In the *synchronous* setting (Ssynch), also called semi-synchronous and first studied in [27], time is divided into discrete intervals, called *rounds*; in each round some robots are activated simultaneously, and perform their LCM cycle in perfect synchronization.

A popular synchronous setting which plays an important role is the *fully-synchronous* setting (Fsynch) where every robot is activated in every round; that is, the activation scheduler has no adversarial power.

[3] If (strong) multiplicity detection is assumed, the snapshot is a multi-set.

In all two settings, the selection of which robots are activated at a round is made by an adversarial *scheduler*, whose only limit is that every robot must be activated infinitely often (i.e., it is fair scheduler). In the following, for all synchronous schedulers, we use round and time interchangeably.

2.4 Computational Relationships

Let $\mathcal{M} = \{\mathcal{LUMI}, \mathcal{FCOM}, \mathcal{FSTA}, \mathcal{OBLOT}\}$ be the set of models under investigation, and $\mathcal{S} = \{\text{FSYNCH}, \text{SSYNCH}\}$ be the set of activation schedulers under consideration.

We denote by $\mathcal{R}$ the set of all teams of robots satisfying the core assumptions (i.e., they are identical, autonomous, and operate in LCM cycles), and $R \in \mathcal{R}$ a team of robots having identical capabilities (e.g., common coordinate system, persistent storage, internal identity, rigid movements etc.). By $\mathcal{R}_n \subset \mathcal{R}$ we denote the set of all teams of size n.

Given a model $M \in \mathcal{M}$, a scheduler $S \in \mathcal{S}$, and a team of robots $R \in \mathcal{R}$, let $Task(M, S; R)$ denote the set of problems solvable by R in M under adversarial scheduler S.

Let $M_1, M_2 \in \mathcal{M}$ and $S_1, S_2 \in \mathcal{S}$. We define the following relationships between model M_1 under scheduler S_1 and model M_2 under scheduler S_2:

- *computationally not less powerful* ($M_1^{S_1} \geq M_2^{S_2}$), if $\forall R \in \mathcal{R}$ we have $Task(M_1, S_1; R) \supseteq Task(M_2, S_2; R)$;
- *computationally more powerful* ($M_1^{S_1} > M_2^{S_2}$), if $M_1^{S_1} \geq M_2^{S_2}$ and $\exists R \in \mathcal{R}$ such that $Task(M_1, S_1; R) \setminus Task(M_2, S_2; R) \neq \emptyset$;
- *computationally equivalent* ($M_1^{S_1} \equiv M_2^{S_2}$), if $M_1^{S_1} \geq M_2^{S_2}$ and $M_2^{S_2} \geq M_1^{S_1}$;
- *computationally orthogonal* (or *incomparable*), ($M_1^{S_1} \perp M_2^{S_2}$), if $\exists R_1, R_2 \in \mathcal{R}$ such that $Task(M_1, S_1; R_1) \setminus Task(M_2, S_2; R_1) \neq \emptyset$ and $Task(M_2, S_2; R_2) \setminus Task(M_1, S_1; R_2) \neq \emptyset$.

For simplicity of notation, for a model $M \in \mathcal{M}$, let M^F and M^S denote M^{Fsynch} and M^{Ssync}, respectively; and let $M^F(R)$ and $M^S(R)$ denote $Task(M,\text{FSYNCH};R)$ and $Task(M,\text{SSYNCH};R)$, respectively.

Trivially, for any $M \in \mathcal{M}$, $M^F \geq M^S$; also, for any $S \in \mathcal{S}$, $\mathcal{LUMI}^S \geq \mathcal{FSTA}^S \geq \mathcal{OBLOT}^S$ and $\mathcal{LUMI}^S \geq \mathcal{FCOM}^S \geq \mathcal{OBLOT}^S$.

3 COMPUTATIONAL RELATIONSHIP IN Fsynch

In this section, we consider the fully synchronous scheduler FSYNCH and we prove that, in this setting, it is better to communicate than to remember. Specifically, we prove that $\mathcal{FCOM}$ has the same power as $\mathcal{LUMI}$ and is strictly more powerful than $\mathcal{FSTA}$; furthermore, they are all strictly more powerful than $\mathcal{OBLOT}$.

3.1 $\mathcal{FCOM}^F \equiv \mathcal{LUMI}^F$

To prove that $\mathcal{FCOM}$ has the same power as $\mathcal{LUMI}$ in FSYNCH, we first need to prove the following.

▶ **Lemma 1.** $\forall R \in \mathcal{R}, \mathcal{LUMI}^F(R) \subseteq \mathcal{FCOM}^F(R)$.

Proof. The proof is constructive. Our algorithm uses the following observation: if there is chirality, then there exists a unique circular ordering of the locations $X(t)$ occupied by the robots at that time [27]. Let `suc` and `pred` be the functions denoting the ordering and, without loss of generality, let $\texttt{suc}(x_i(t)) = x_{i+1 \bmod m}(t)$ and $\texttt{pred}(x_i)(t) = x_{i-1 \bmod m}(t)$ for $i \in \{0, 1, \ldots, m-1\}$. Even in absence of chirality, a circular arrangement can still be obtained,

but there is no common agreement on `suc` and `pred` because the "clockwise" direction is not common to all robots and the notion of successor and predecessor is local, and possibly inconsistent among the robots. In this case, let $\texttt{neigh}(x_i(t))$ indicate the unordered pair of the two neighbouring locations of x_i: $\texttt{neigh}(x_i(t)) = \{x_{i+1 \bmod m}(t), x_{i-1 \bmod m}(t)\}$ for $i \in \{0, 1, \ldots, m-1\}$. When no ambiguity arises, we will omit the temporal indication.

We now describe an $\mathcal{FCOM}$ protocol, called `LUbyFCinFSY`, which, for any given $\mathcal{LUMI}$ protocol A, produces a fully-synchronous execution of A. The simulation algorithm is presented in Algorithm 1, where a robot r at location x uses three lights: $r.color$, indicating its own color, initially set to c_0, $r.\texttt{neigh}.color$, indicating the 2-element set of colors seen at $\texttt{suc}(x)$ and at $\texttt{pred}(x)$ taken from the set $2^{\mathcal{C}}$, where C is the set of colors used by algorithm A, initially set to $\{\{c_0\}, \{c_0\}\}$, and $r.step \in \{1, 2\}$, indicating the step of the algorithm, initially set to 1. It also uses variable $r.color.here$, initially set to $\{c_0\}$, indicating the set of colors visible by r at its own location. In the following, when no ambiguity arises, we will denote $\texttt{suc}(x)$ and $\texttt{pred}(x)$ by $\texttt{suc}(r)$ and $\texttt{pred}(r)$.

The algorithm simulates a single round of A with two rounds (or steps):

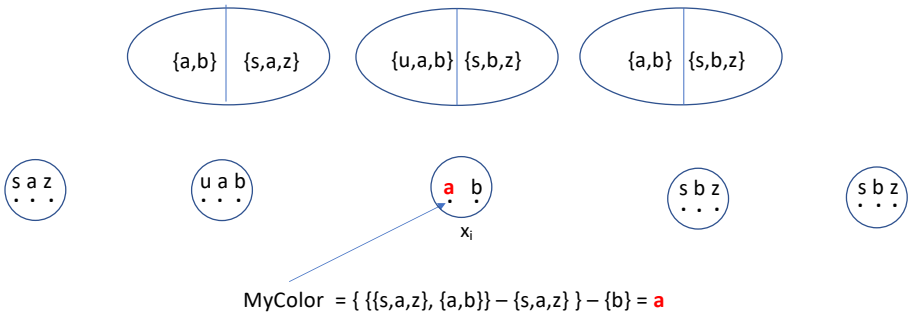


Figure 1 $\{\texttt{pred}(x).\texttt{neigh}.color - r'.color\} - r.color.here$.

1. **Copy Step:** $(r.step = 1)$. In the Look phase, r determines $r.step = 1$ by observing the corresponding color of one of the neighbours (e.g., $\texttt{pred}(x).step$) and sets $r.step = 2$. It also observes the colors of the robots at its successor and predecessor and sets $r.\texttt{neigh}.color$ (notice that $r.\texttt{neigh}.color$ is the same for all robots at the same location). Robot r does not move.
2. **Execution Step:** $(r.step = 2)$.
   Color Determination. After the Look phase, by looking at one of its neighbours ($\texttt{pred}(x)$) robot r discovers $r.step = 2$, as well as its own color. In fact, let $x' = \texttt{other}(\texttt{pred}(x))$ denote the other neighbour of r's predecessor, and let $r.color.here$ correspond to the set of colors seen by r at its own location x (note that, by definition, this set does not include r's color); then r's color is determined by letting *cand-set* be the element of $\texttt{pred}(x).\texttt{neigh}.color - \{x'.color\}$ and r's color be the element of $cand\text{-}set - r.color.here$, where "-" indicates the difference operator between sets (see Figure 1).
   Execution. Robot r executes the `Compute` and `Move` phases according to Algorithm A.

The correctness of Algorithm $\texttt{LUbyFCinFSY}(A)$ follows easily from the fact that we are operating in FSYNCH and that the only difference between $\mathcal{LUMI}$ and $\mathcal{FCOM}$ is that in latter a robot does not see the color of its own light. This can however be determined as indicated in the protocol. In other words, $\texttt{LUbyFCinFSY}(A)$ correctly simulate in FSYNCH algorithm A and Lemma 1 follows. ◀

Algorithm 1 `LUbyFCinFSY`(A) - for robot r at location x.

Phase Look
Observe, in particular, $\texttt{pred}(x).color$, $\texttt{suc}(x).color$, $\texttt{pred}(x).step$, $\texttt{other}(\texttt{pred}(x))$;
as well as $r.color.here$ (note that, for this, r cannot see its own color).

Phase Compute
if $(\texttt{pred}(x).step = 1)$ **then** //step 1- Copy //
  $r.\texttt{neigh}.color \leftarrow \{\texttt{pred}(x).color, \texttt{suc}(x).color\}$,
  where $\texttt{pred}$(x).color=$\{\rho.color | \rho \in \texttt{pred}(x)\}$ and $\texttt{suc}$(x).color=$\{\rho.color | \rho \in \texttt{suc}(x)\}$
  $r.step \leftarrow 2$
  $r.des \leftarrow x$
else //step 2- Execution //
  $x' \leftarrow \texttt{other}(\texttt{pred}(x))$ // x' is the other neighbour of $\texttt{pred}(x)$ //
  $cand\text{-}set \leftarrow$ the element of $\texttt{pred}(x).\texttt{neigh}.color - \{x'.color\}$
  $r.color \leftarrow$ the element of $cand\text{-}set - r.color.here$ // find my own color //
  Execute the `Compute` of $\mathcal{A}$ // with my color $r.color$, determining destination $r.des$ //

Phase Move
Move to $r.des$;

Since the reverse relation $\mathcal{FCOM}^F \leq \mathcal{LUMI}^F$ holds by definition, we can conclude:

▶ **Theorem 2.** $\mathcal{FCOM}^F \equiv \mathcal{LUMI}^F$.

3.2 $\mathcal{FCOM}^F > \mathcal{FSTA}^F$

We now turn our attention to the relationship between $\mathcal{FCOM}^{\mathcal{F}}$ and $\mathcal{FSTA}^F$. The following problem is used to show that $\mathcal{FCOM}^F > \mathcal{FSTA}^F$.

▶ **Definition 3.** ***Problem ¬IL****: Three robots a, b, and c, starting from the initial configuration shown in Figure 2 (a), must form first the pattern of Figure 2 (b) and then move to form the pattern of Figure 2 (c).*

▶ **Lemma 4.** $\exists R \in \mathcal{R}_3$, $\neg IL \notin \mathcal{FSTA}^F(R)$,

Proof. In the initial pattern (a) of Figure 2, even if all the states of the robots are initially identical, each of them can uniquely distinguish its position in the pattern. Therefore, the three robots can easily form pattern (b) by having a move clockwise of 90 degrees. Assume that in pattern (b) the state of each robot is now different and indicates the full history of what the robot has done so far. Now the robots need to form pattern (c), which is asymmetric and requires b to move clockwise of 45 degrees. However, in pattern (b), even in presence of chirality, robot b cannot distinguish between the positions of a and c. This is true regardless of the information stored in the local state of robot b; so, after forming pattern (b), the robots cannot reach pattern (c). ◀

▶ **Lemma 5.** $\forall R \in \mathcal{R}_3$, $\neg IL \in \mathcal{FCOM}^S(R)$.

Proof. $\mathcal{FCOM}$ robots can easily solve ¬IL as follows: To form (b) from (a), robot a, which can easily distinguish its position, moves of 90 degrees clockwise and turns its light to `red`. To move from (b) to (c) robot b distinguishes a from c because of the external light and moves of 45 degrees clockwise to occupy the correct position. ◀

By Theorem 2 and Lemmas 4 and 5, we can conclude:

▶ **Theorem 6.** $\mathcal{FCOM}^F > \mathcal{FSTA}^F$.

3.3 $\mathcal{FSTA}^F > \mathcal{OBLOT}^F$

It is very easy to show that $\mathcal{FSTA}$ is strictly more powerful than $\mathcal{OBLI}$. To do that, we consider the *Oscillating Point Problem* defined in [10]

▶ **Definition 7.** ***Problem OSP (Oscillating Points) [10]:*** *Two robots, a and b, initially in distinct locations, alternately come closer and move further from each other. More precisely, let $d(t)$ denote the distance of the two robots at time t. The OSP problem requires the two robots, starting from an arbitrary distance $d(t_0) > 0$ at time t_0, to move so that there exists a monotonically increasing infinite sequence time instant $t_0, t_1, t_2, \ldots$ such that :*

1. $d(t_{2i+1}) < d(t_{2i})$, *and* $\forall h', h" \in [t_{2i}, t_{2i+1}], h' < h", d(h") \leq d(h')$; *and*
2. $d(t_{2i}) > d(t_{2i-1})$, *and* $\forall h', h" \in [t_{2i-1}, t_{2i}], h' < h", d(h") \geq d(h')$.

Impossibility in $\mathcal{OBLOT}^F$ has been shown in [10]:

▶ **Lemma 8.** *[10]* $\exists R \in \mathcal{R}_2$, $OSP \notin \mathcal{OBLOT}^F(R)$.

On the other hand, possibility in $\mathcal{FSTA}^F$ is trivial because a robot can store in its local state whether in the previous round it was moving further or closer and successfully alternate movements. That is

▶ **Lemma 9.** $\forall R \in \mathcal{R}_2$, $OSP \in \mathcal{FSTA}^F(R)$.

By Lemmas 8 and 9, and the fact that $\mathcal{FSTA}^F \geq \mathcal{OBLOT}^F$ by definition, we have:

▶ **Theorem 10.** $\mathcal{FSTA}^F > \mathcal{OBLOT}^F$.

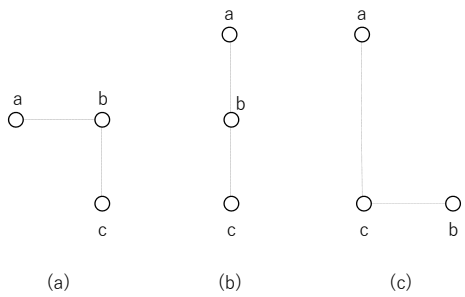


Figure 2 The configurations of problem ¬IL.

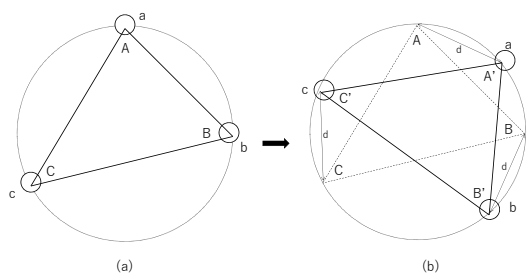


Figure 3 Illustration of TRIANGLE-ROTATION (TAR(d)).

4 COMPUTATIONAL RELATIONSHIP IN Ssynch

In this section, we examine the computational relationship of the models under the Semi-Synchronous scheduler.

4.1 Orthogonality of $\mathcal{FSTA}^S$ and $\mathcal{FCOM}^S$

▶ **Definition 11.** ***Problem TAR(d) (Triangle Rotation):*** *Let a, b, c be three robots forming a triangle ABC, let $\mathcal{C}$ be the circumscribed circle, and let d be a value known to the three robots. The TAR(d) problem requires the robots to move so to form a new triangle $A'B'C'$ with circumscribed circle $\mathcal{C}$, and where $dis(A, A') = dis(B, B') = dis(C, C') = d$ (see Figure 3).*

▶ **Lemma 12.** *$\exists R \in \mathcal{R}_3$, TAR(d) $\notin \mathcal{FCOM}^S(R)$.*

Proof. (Sketch) By contradiction, let $\mathbf{A}$ be a correct solution protocol in $\mathcal{FCOM}^S$. Consider an initial configuration C_0 where the three robots a, b, and c, form a scalene triangle ABC with $AB \neq d$, $BC \neq d$, $CA \neq d$, and with all lights `off` (see Figure 3(a)). Consider now an execution $\mathcal{E}$ of $\mathbf{A}$ where all three robots are activated in each round, starting from C_0, until one or more robots move, say at round k. Let r be a robot that performed a non-null move in that round after observing configuration C_{k-1}. Consider now another execution $\mathcal{E}'$ of $\mathbf{A}$ where the first $k-1$ rounds are exactly the same, but in round k robot r is the only one activated. Robot r would move to a new location possibly changing color. Now the schedule activates again only robot r. If the previous move resulted in a scalene triangle, the robot cannot distinguish this situation from the one it observed at the previous round and thus it would perform the same type of movement, losing any information on the original triangle; if the previous move resulted in an equilateral or isosceles triangle, robot r would know it has already moved (even without having access to its light), but it still would not know from which location. In both cases the information on the original triangle cannot be reconstructed and the problem cannot be solved, contradicting the correctness of $\mathbf{A}$. ◀

▶ **Lemma 13.** *$\forall R \in \mathcal{R}_3$, TAR(d) $\in \mathcal{FSTA}^S(R)$.*

Proof. The problem is easily solvable with $\mathcal{FSTA}$ robots in Ssynch. Let the robots have color A initially. The first time a robot is activated, it moves to the desired position and changes its light to B. Whenever a robot is activated, if its light is B, it does not move. ◀

By Lemmas 4-5 and 12-13, we can conclude:

▶ **Theorem 14.** *$\mathcal{FCOM}^S \perp \mathcal{FSTA}^S$.*

4.2 Dominance of $\mathcal{FSTA}^S$ and $\mathcal{FCOM}^S$ over $\mathcal{OBLOT}^S$

The dominance of $\mathcal{FSTA}^S$ and $\mathcal{FCOM}^S$ over $\mathcal{OBLOT}^S$ follows directly from existing results on the rendezvous problem (RDV), which prescribes two robots to occupy exactly the same location, not known in advance.

▶ **Theorem 15.** *$\mathcal{FSTA}^S > \mathcal{OBLOT}^S$ and $\mathcal{FCOM}^S > \mathcal{OBLOT}^S$.*

Proof. It is well known that RDV cannot be solved in Ssynch (see [27], whose proof uses chirality and trivially holds when movements are rigid). On the other hand, it can be solved in $\mathcal{FCOM}$ and $\mathcal{FSTA}$ in Ssynch [17]. ◀

4.3 Dominance of $\mathcal{LUMI}^S$ over $\mathcal{FSTA}^S$ and $\mathcal{FCOM}^S$

To conclude the study of Ssynch, we consider the OSP problem already employed in Section 3.3. also to show that $\mathcal{LUMI}^S > \mathcal{FSTA}^S(\mathcal{FCOM}^S)$.

▶ **Lemma 16.**
- $\exists R \in \mathcal{R}_2,\ OSP \notin \mathcal{FCOM}^S(R) \cup \mathcal{FSTA}^S(R)$.
- $\forall R \in \mathcal{R}_2,\ OSP \in LUMI^S(R)$.

Proof. The possibility in $\mathcal{LUMI}^S$ is proven in [10]. Let us then prove the impossibility in $\mathcal{FCOM}$ and $\mathcal{FSTA}$. Let a and b be the two robots with initial lights `off`. First note that if an activated robot performs a null move at the first round, the adversarial scheduler would activate both (making them change lights in the same way). The scheduler continues to activate them both until the first round t when the color of the light would make them do a non-null move. At this point, the scheduler changes strategy.

In the case of $\mathcal{FCOM}$, the scheduler activates only robot a in the two consecutive rounds t and $t+1$. At round $t+2$, robot a is activated again. Robot a will repeat (incorrectly) the same move at round $t+2$, not being able to distinguish the current situation from the previous, and regardless of the movement taken in round t.

In the case of $\mathcal{FSTA}$, the scheduler activates only robot a for 3 consecutive rounds $t, t+1, t+2$ and both robots at round $t+3$. In the first 3 activations robot a can use its internal light to correctly alternate a move going closer to b, one moving further and the third moving closer again. At round $t+3$, robot a will necessarily move further from b continuing this alternating pattern (as nothing has changed in its perceived view of the universe), but robot b is now in the same state robot a was at round t and will therefore take the same action taken by a at that round (i.e., moving closer to a). This lack of synchronization makes the robots incorrectly maintain their distance during round $t+3$. ◀

We can conclude that:

▶ **Theorem 17.** *$\mathcal{LUMI}^S > \mathcal{FSTA}^S$ and $\mathcal{LUMI}^S > \mathcal{FCOM}^S$.*

5 COMPUTATIONAL RELATIONSHIP BETWEEN Fsynch AND Ssynch

In this section we examine the computational relationship of fully synchronous and semi-synchronous models.

5.1 Dominances of Fsynch over Ssynch

The following problem prescribes the robots to perform a sort of "expansion" of the initial configuration with respect to their center of gravity; specifically, each robot must move away from the center of gravity (c_x, c_y) to the closest integral position corresponding to doubling its distance from it. More precisely:

▶ **Definition 18.** ***Problem CGE (Center of Gravity Expansion)****: Let R be a set of robots. The CGE problem requires each robot $r_i \in R$ to move from its initial position (x_i, y_i) directly to $(f(x_i, c_x), f(y_i, c_y))$, where $f(a, b) = \lfloor 2a - b \rfloor$ and (c_x, c_y) is the center of gravity of the initial configuration.*

▶ **Lemma 19.** *$CGE \in \mathcal{FSTA}^F$ and $CGE \notin \mathcal{LUMI}^S$.*

Proof. (Sketch) It is easy to see that $CGE \in \mathcal{FSTA}^F$ since all robots can simultaneously reach their destination in one step and change color to indicate termination. We now show that $CGE \notin \mathcal{LUMI}^S$. By contradiction. Consider an execution $\mathcal{E}$ of a solution algorithm where a single robot r is activated at the first time step. The robot moves correctly to its

destination point and possibly changes its color. After this movement, regardless of the distance traveled, the center of gravity of the new configuration is different from the one of the initial configuration, with respect to which all the other robots must move. At the next activation, any robot different from r must move to its target location; however, this cannot be done because the robot cannot reconstruct the exact position of the original center of gravity. This is due to the fact that there are infinite combinations of coordinates from where r could have feasibly moved and the reconstruction of the original CoG cannot be done just on the basis of a light that can carry finite information. ◀

As a consequence, we have that:

▶ **Theorem 20.**

1. $\mathcal{LUMI}^F > \mathcal{LUMI}^S$
2. $\mathcal{FSTA}^F > \mathcal{FSTA}^S$
3. $\mathcal{FCOM}^F > \mathcal{LUMI}^S > \mathcal{FCOM}^S$
4. $\mathcal{OBLOT}^F > \mathcal{OBLOT}^S$

Proof.

1. It follows from Lemma 19, Theorem 2, and Theorem 6.
2. It follows from Lemma 19 and Theorem 17.
3. It follows immediately from Theorem 2, Theorem 17, and Theorem 20.
4. The RDV problem can be trivially solved in $\mathcal{OBLOT}^F$ but it cannot be solved in $\mathcal{OBLOT}^S$[27]. ◀

5.2 Incomparabilities between Fsynch and Ssynch

5.2.1 Orthogonality of $\mathcal{OBLOT}^F$ with $\mathcal{FCOM}^S$ and $\mathcal{FSTA}^S$

Consider the following problem:

▶ **Definition 21.** ***Problem SRO (Shrinking Rotation)****: Two robots a and b are initially placed in arbitrary distinct points (forming the initial configuration C_0), The two robots uniquely identify a square (initially Q_0) whose diagonal is given by the segment between them*[4]*. Let a_0 and b_0 indicate the initial positions of the robots, d_0 the segment between them, and $length(d_0)$ its length. Let a_i and b_i be the positions of a and b in configuration C_i ($i \geq 0$). The problem consists of moving from configuration C_i to C_{i+1} in such a way that Condition **C3** is verified and so is one of **C1** and **C2**:*

C1. *d_{i+1} is a 90 degree clockwise rotation of d_i and thus $length(d_{i+1}) = length(d_i)$,*

C2. *d_{i+1} is a "shrunken" 45 degree clockwise rotation of d_i such that $d_{i+1} = \frac{d_i}{\sqrt{2}}$,*

C3. *a_{i+1} and b_{i+1} must be included in the square Q_{i-1}, where Q_{-1} is the infinite square.*

[4] By square, we means the entire space delimited by the four sides.

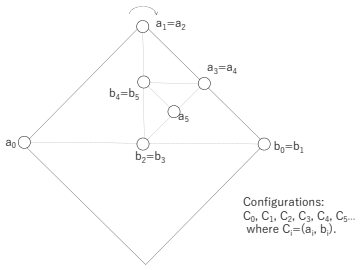


Figure 4 Illustration of SHRINKING ROTATION (SRO).

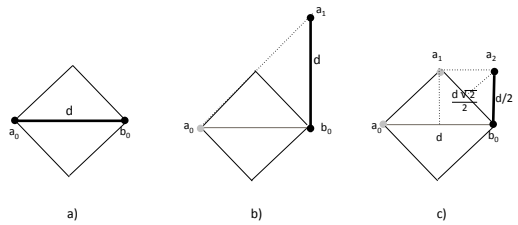


Figure 5 Proof of Lemma 23: a) Initial configuration; b) after the movement of robot a in Case (1); c) after two consecutive movements of robot a in Case (2).

▶ **Lemma 22.** $\forall R \in \mathcal{R}_2, SRO \in \mathcal{OBLOT}^F(R)$

Proof. The proof is by construction: Each robot rotates clockwise of 90 degrees with respect to the midpoint between itself and the other robot. Since the schedule is FSYNCH, it allows consecutive simultaneous activation of the two robots. So, there is only one possible type of executions under FSYNCH with two robots: a perpetual activation of both robots in each round. In this case, the problem is clearly solved by the algorithm stated above, because the robots keep rotating of 90 degrees clockwise around their mid-point, fulfilling **C1** and **C3**. Note that **C2** never happens under FSYNCH. Then SRO can be solved with $\mathcal{OBLOT}$ in FSYNCH. ◀

▶ **Lemma 23.** $\exists R \in \mathcal{R}_2, SRO \notin \mathcal{FCOM}^S(R) \cup \mathcal{FSTA}^S(R)$

Proof. First note that if an activated robot performs a null move at the first round, the schedule would activate both (making them change lights in the same way). The scheduler continues to activate them both until the first round i when the color of the light would make them do a non-null move. At this point, the scheduler changes strategy.

Consider first the case of $\mathcal{FCOM}^S$ and consider an execution where a robot, say a, is activated (alone) twice consecutively starting from configuration C_i. In the following, we show that, under this activation schedule, either C_{i+1} or C_{i+2} would violate **C3** (which states that a_{i+1} and b_{i+1} must be included in the square Q_{i-1}) (see Figure 4).
In fact, let robot a located at a_i be activated from a configuration C_i. Since b is not activated in C_i, the light of b at b_i and at b_{i+1} are the same. Then a at a_i and at a_{i+1} observe the same light on b. Since the coordinate systems of the robot can be chosen so that they have the same view of the universe, a at a_{i+1} performs the same action as it would perform at a_i, and this action must either fulfill **C1** or **C2** (as well as **C3** in either case).
Case (1). Let us consider first the situation when **C1** is fulfilled with a single movement of a: the only possibility would be for a to rotate clockwise of 90 degree with respect to b; this movement, however, would immediately violate **C3** because the new position a_{i+1} would be outside of the square Q_i (and thus also outside Q_{i-1}) (see Figure 5 from a) to b)).
Case (2). Let us consider now the case when **C2** is fulfilled with a single movement of a: the only possibility would be for a to move clockwise of 90 degrees with respect to the midpoint between a and b reaching a feasible configuration C_{i+1}. When robot a is activated again at the next round, it will perform the same action on C_{i+1}, now violating **C3** (see Figure 5 from a) to c)).

Therefore, this problem cannot be solved with $\mathcal{FCOM}$ in SSYNCH. The case of $\mathcal{FSTA}^S$ can be shown in a similar way, because the availability of internal lights cannot prevent - in SSYNCH - the consecutive activation of the same single robot and the impossibility argument described above would still hold. ◀

Moreover, we have:

▶ **Lemma 24.** $\forall R \in \mathcal{R}_2, SRO \in \mathcal{LUMI}^S(R)$

Proof. It is rather straightforward to see that in $\mathcal{LUMI}^S$ the two robots can be synchronized with 3 colors so to enforce a fully synchronous execution. ◀

We have seen that SRO can be solved in $\mathcal{OBLOT}^F$ but cannot be solved in $\mathcal{FCOM}^S$ and $\mathcal{FSTA}^S$. On the other hand, $\neg IL$ and TAR(d) can be solved in $\mathcal{FCOM}^S$ and $\mathcal{FSTA}^S$, respectively, but cannot be solved in $\mathcal{OBLOT}^F$. We can conclude that:

▶ **Theorem 25.** *$\mathcal{OBLOT}^F \perp \mathcal{FCOM}^S$ and $\mathcal{OBLOT}^F \perp \mathcal{FSTA}^S$.*

5.2.2 Orthogonality of $\mathcal{LUMI}^S$ with $\mathcal{FSTA}^F$ and $\mathcal{OBLOT}^F$

▶ **Theorem 26.** *$\mathcal{LUMI}^S \perp \mathcal{FSTA}^F$ and $\mathcal{FCOM}^S \perp \mathcal{FSTA}^F$.*

Proof. Problem ¬IL can be solved in $\mathcal{FCOM}^S$ (and thus in $\mathcal{LUMI}^S$) but not in $\mathcal{FSTA}^F$ (Lemmas 4 and 5). Problem CGE can be solved in $\mathcal{FSTA}^F$, but not in $\mathcal{LUMI}^S$ (Lemma 19). ◀

▶ **Definition 27.** ***Problem CGE* (Perpetual Center of Gravity Expansion).*** *This is the same as CGE, where however after each expansion, the robots have to repeat the same process from the new configuration.*

▶ **Theorem 28.** *$\mathcal{LUMI}^S \perp \mathcal{OBLOT}^F$.*

Proof. Problem OSP can be solved in $\mathcal{LUMI}^S$ (Lemma 16), but not in $\mathcal{OBLOT}^F$ (Lemma 8). Problem CoG* can be trivially solved in $\mathcal{OBLOT}^F$, but not in $\mathcal{LUMI}^S$ (Lemma 19). ◀

Let us remark that, since $\mathcal{LUMI}^s \equiv \mathcal{LUMI}^A$, the result of Theorem 28 answers the open question on the relationship between $\mathcal{LUMI}^A$ and $\mathcal{OBLOT}^F$ posed in [10].

6 CONCLUDING REMARKS

In this paper, we have investigated the computational power of communication versus persistent memory in mobile robots by studying the relationship among $\mathcal{LUMI}$, $\mathcal{FCOM}$, $\mathcal{FSTA}$ and $\mathcal{OBLOT}$ models, and we have shown that their relationship depends of the scheduler under which the robots operate. We considered the two classical synchronous schedulers, FSYNCH and SSYNCH, establishing several results. In particular, we proved that communication is more powerful than persistent memory if the scheduler is fully synchronous; on the other hand, the two models are incomparable under the semi-synchronous scheduler. For an overall panorama of the established relationship among the models, see Figure 6.

Several problems are still open. An outstanding open problem is the study of the relationship among these models in ASYNCH, where there is no notion of rounds and the cycles of the robots are executed independently.

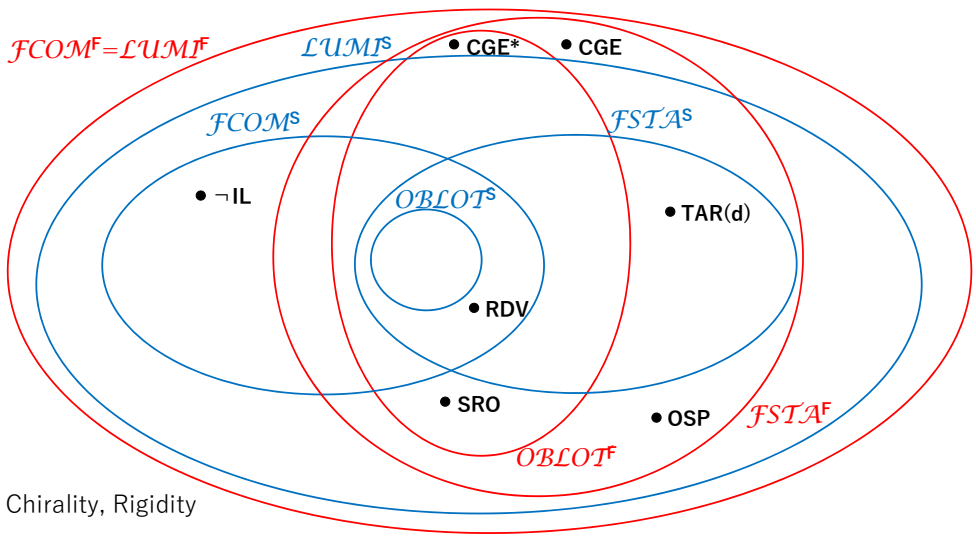


Figure 6 Relationship among $\mathcal{LUMI}$, $\mathcal{FCOM}$, $\mathcal{FSTA}$ and $\mathcal{OBLOT}$ in Fsynch, and Ssynch assuming chirality and rigidity.

Another open problem is whether there exists a scheduler S' ("weaker" than Fsynch but stronger than Ssynch) such that each model under S' would be computationally equivalent to the same model under Fsynch.

Finally, most of the results of this paper hold assuming chirality and rigidity (exceptions are the RDV-algorithms, the OSP-algorithms, and the simulation algorithm, Algorithm 1, which do not require either). It is an open question to characterize the inclusions among all the various models in the case of disoriented robots with non-rigid movement.

References

1. N. Agmon and D. Peleg. Fault-tolerant gathering algorithms for autonomous mobile robots. *SIAM Journal on Computing*, 36(1):56–82, 2006.
2. H. Ando, Y. Osawa, I. Suzuki, and M. Yamashita. A distributed memoryless point concergence algorithm for mobile robots with limited visivility. *IEEE Transactions on Robotics and Automation*, 15(5):818–828, 1999.
3. S. Bhagat and K. Mukhopadhyaya. Optimum algorithm for mutual visibility among asynchronous robots with lights. In *Proc. of the 19th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 341–355, 2017.
4. Z. Bouzid, S. Das, and S. Tixeuil. Gathering of mobile robots tolerating multiple crash Faults. In *the 33rd Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 334–346, 2013.
5. D. Canepa and M. Gradinariu Potop-Butucaru. Stabilizing flocking via leader election in robot networks. In *Proc. of the 10th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 52–66, 2007.
6. S. Cicerone, Di Stefano, and A. Navarra. Gathering of robots on meeting-points. *Distributed Computing*, 31(1):1–50, 2018.

7 M. Cieliebak, P. Flocchini, G. Prencipe, and N. Santoro. Distributed computing by mobile robots: Gathering. *sicomp*, 41(4):829–879, 2012.

8 R. Cohen and D. Peleg. Convergence properties of the gravitational algorithms in asynchronous robot systems. *sicomp*, 34(15):1516–1528, 2005.

9 S. Das, P. Flocchini, G. Prencipe, N. Santoro, and M. Yamashita. The power of lights: synchronizing asynchronous robots using visible bits. In *Proc. of the 32nd International Conference on Distributed Computing Systems (ICDCS)*, pages 506–515, 2012.

10 S. Das, P. Flocchini, G. Prencipe, N. Santoro, and M. Yamashita. Autonomous mobile robots with lights. *Theoretical Computer Science*, 609:171–184, 2016.

11 G.A. Di Luna, P. Flocchini, S.G. Chaudhuri, F. Poloni, N. Santoro, and G. Viglietta. Mutual visibility by luminous robots without collisions. *Information and Computation*, 254(3):392–418, 2017.

12 G.A. Di Luna and G. Viglietta. *Robots with Lights*. Chapter 11 of [13], pages 252–277, 2019.

13 P. Flocchini, G. Prencipe, and N. Santoro (Eds). *Distributed Computing by Mobile Entities*. Springer, 2019.

14 P. Flocchini, G. Prencipe, N. Santoro, and P. Widmayer. Hard tasks for weak robots: the role of common knowledge in pattern formation by autonomous mobile robots. In *Proc. of 10th International Symposium on Algorithms and Computation (ISAAC)*, pages 93–102, 1999.

15 P. Flocchini, G. Prencipe, N. Santoro, and P. Widmayer. Gathering of asynchronous robots with limited visibility. *tcs*, 337(1–3):147–169, 2005.

16 P. Flocchini, G. Prencipe, N. Santoro, and P. Widmayer. Arbitrary pattern formation by asynchronous oblivious robots. *tcs*, 407:412–447, 2008.

17 P. Flocchini, N. Santoro, G. Viglietta, and M. Yamashita. Rendezvous with Constant Memory. *tcs*, 621:57–72, 2016.

18 N. Fujinaga, Y. Yamauchi, H. Ono, S. Kijima, and M. Yamashita. Pattern formation by oblivious asynchronous mobile robots. *sicomp*, 44(3):740–785, 2015.

19 V. Gervasi and G. Prencipe. Coordination without communication: The case of the flocking problem. *Discrete Applied Mathematics*, 144(3):324–344, 2004.

20 A. Hériban, X. Défago, and S. Tixeuil. Optimally gathering two robots. In *Proc. of the 19th Int. Conference on Distributed Computing and Networking (ICDCN)*, pages 1–10, 2018.

21 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.

22 T. Okumura, K. Wada, and X. Défago. Optimal Rendezvous $\mathcal{L}$-Algorithms for Asynchronous Mobile Robots with External-Lights. In *Proc. of the 22nd Int. Conference on Principles of Distributed Systems (OPODIS)*, pages 24:1–24:16, 2018.

23 T. Okumura, K. Wada, and Y. Katayama. Brief Announcement: Optimal asynchronous Rendezvous for mobile robots with lights. In *Proc. of the 19th Int. Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 484–488, 2017.

24 D. Peleg. Distributed coordination algorithms for mobile robot swarms: New directions and challenges. In *Proc. of 7th International Workshop on Distributed Computing (IWDC)*, pages 1–12, 2005.

25 G. Sharma, R. Alsaedi, C. Bush, and S. Mukhopadyay. The complete visibility problem for fat robots with lights. In *Proc. of the 19th International Conference on Distributed Computing and Networking (ICDCN)*, pages 21:1–21:4, 2018.

26 S. Souissi, T. Izumi, and K. Wada. Oracle-based flocking of mobile robots in crash-recovery model. In *Proc. of the 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 683–697, 2009.

27 I. Suzuki and M. Yamashita. Distributed anonymous mobile robots: Formation of geometric patterns. *sicomp*, 28:1347–1363, 1999.

28 S. Terai, K. Wada, and Y. Katayama. Gathering problems for autonomous mobile robots with lights. *arXiv.org*, cs(ArXiv:1811.12068), 2018.

29 G. Viglietta. Rendezvous of two robots with visible bits. In *10th International Symposium on Algorithms and Experiments for Sensor Systems, Wireless Networks and Distributed Robotics (ALGOSENSORS)*, pages 291–306, 2013.

30 M. Yamashita and I. Suzuki. Characterizing geometric patterns formable by oblivious anonymous mobile robots. *tcs*, 411(26–28):2433–2453, 2010.

31 Y. Yamauchi, T. Uehara, S. Kijima, and M. Yamashita. Plane formation by synchronous mobile robots in the three-dimensional euclidean space. *Journal of the ACM (JACM)*, 64:3(16):16:1–16:43, 2017.

Lower Bounds for Shoreline Searching With 2 or More Robots

Sumi Acharjee
Department of Mathematics, Ryerson University, Toronto, Canada
sumi.acharjee@ryerson.ca

Konstantinos Georgiou[1]
Department of Mathematics, Ryerson University, Toronto, Canada
konstantinos@ryerson.ca

Somnath Kundu
Department of Mathematics, Ryerson University, Toronto, Canada
somnath.kundu@ryerson.ca

Akshaya Srinivasan
Department of Computer Science & Engineering, National Institute of Technology, Tiruchirappalli, India
akshaya.kms@gmail.com

Abstract

Searching for a line on the plane with n unit speed robots is a classic online problem that dates back to the 50's, and for which competitive ratio upper bounds are known for every $n \geq 1$, see [5]. In this work we improve the best lower bound known for $n = 2$ robots [5] from 1.5993 to 3. Moreover we prove that the competitive ratio is at least $\sqrt{3}$ for $n = 3$ robots, and at least $1/\cos(\pi/n)$ for $n \geq 4$ robots. Our lower bounds match the best upper bounds known for $n \geq 4$, hence resolving these cases. To the best of our knowledge, these are the first lower bounds proven for the cases $n \geq 3$ of this several decades old problem.

2012 ACM Subject Classification Theory of computation → Online algorithms; Theory of computation → Distributed algorithms; Theory of computation → Computational geometry; Computing methodologies → Continuous space search

Keywords and phrases 2-Dimensional Search, Online Algorithms, Competitive Analysis, Lower Bounds

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.26

Funding *Konstantinos Georgiou*: Research supported in part by NSERC.
Akshaya Srinivasan: Research supported in part by MITACS Globalink.

1 Introduction

Searching for a shoreline is the problem in which a number of identical unit speed searchers, starting from the same point on the plane, need to agree on trajectories so as to hit (eventually) any line on the plane. The underlying optimization problem asks for fixed trajectories, one for each searcher, so as to minimize the worst case relative time untill the first searcher hits the line, i.e. the time untill the line is found divided by the distance of the line to the origin. This two-dimensional search-type problem has a long history, and conjectured optimal strategies have been proposed for every $n \geq 1$ (see Section 1.1 for detailed discussion), where n is the number of searchers (robots). Similarly to the much easier one dimensional analog of the problem, known as the cow-path problem, showing competitive ratio lower bounds

[1] Corresponding author

© Sumi Acharjee, Konstantinos Georgiou, Somnath Kundu, and Akshaya Srinivasan;
licensed under Creative Commons License CC-BY
23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller; Article No. 26; pp. 26:1–26:11
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

for the problem has been a much more challenging task. Indeed, for the shoreline problem, very weak unconditional lower bounds are known for $n = 1$, while the only non-trivial lower bounds known for other values of n is that for $n = 2$.

In this work we improve the state-of-the-art when it comes to competitive ratio lower bounds for searching for a shoreline with $n \geq 2$ robots. In particular, we improve the best lower bound known for $n = 2$ robots, from 1.5993 to 3. Then, we prove the first lower bounds for $n \geq 3$ robots. More specifically, we show a lower bound of $\sqrt{3}$ for 3 robots, and $1/\cos(\pi/n)$ for $n \geq 4$ robots, matching this way the best upper bound known for the latter case.

1.1 Related Work

Theory of search has a long history that dates back to the 50's, see [8, 9]. In one of the simplest continuous problems, a unit speed robot is moving on an infinite line, and its goal is to hit every point (bounded away from the origin) within bounded relative time. The problem, now known as linear-search or cow-path, was restudied by the computer science community in the late 80's in [6], and became so fruitful that numerous variations emerged with challenging and particularly interesting algorithmic problems. Indeed over the decades, accumulated results were summarized in a number of interesting surveys, e.g. [10, 13, 14, 19]. Moreover, the underlying mathematical theory became rich enough to give rise to a number of related books, with [1, 2, 3] being the most relevant and influential.

Among the numerous variations/generalizations of the cow-path problem, the current work focuses on its 2-dimensional analog, that we call the $\textsc{Shoreline}_n$ problem, in which n robots are searching in parallel on the plane for a line. As it is outside the scope of this paper to do a thorough literature review on search-type problems, we refer the reader to the aforementioned surveys and books for all remotely related results, and we focus here on the literature closely related to the shoreline problem, i.e. to 2-dimensional search problems with $n \geq 1$ robots. The language that we adopt for quantifying algorithms' performance is that of competitive analysis, e.g. see [11]. In particular, we think of our problem as n robots, starting from the origin, that are solving an online problem in which a line is placed at an unknown location δ away from the origin, where in particular δ is unknown (but it is bounded away from 0, and that bound is known). The goal of the search is to minimize the relative worst case search time, i.e. the time untill the line is found by any robot divided by δ, over all possible placements of lines and over all δ. The best possible relative time is known as the *competitive ratio* of the problem, and can be thought as the best worst case relative performance of an online algorithm (that does not know the input) compared to the performance of the best offline algorithm (that knows the input).

Searching for a (shore)line with 1 robot, without any knowledge of its distance to the origin, was first proposed in [6], and a number of improvements were proposed for parallel search in [4, 5, 7, 23], i.e. for $n \geq 2$ robots and a number of variations. The best algorithm known for $\textsc{Shoreline}_1$ is a logarithmic spiral search that has competitive ratio 13.81 [6]. Notably, the only unconditional lower bound for the problem is that the competitive ratio is at least 6.3972 [5] which also holds true if the online algorithm knows, a priori, the distance of the line to the origin. Only assuming a cyclic-type trajectory, the competitive ratio is provably at least 12.5385 [25].

The overall picture for searching with $n \geq 2$ robots for a line (without any knowledge about the hidden line) is much more blurry. For $n = 2$, a double logarithmic spiral, in which the origin lies always in the middle of the locations of the robots is known to induce competitive ratio 5.2644 [5]. The only lower bound to the problem is due to the variation

in which the distance is known, and it is 1.5993 [5]. For $n \geq 3$, the natural algorithm of [5] makes robots move along rays, splitting the plane evenly, and induces competitive ratio at most $1/\cos(\pi/n)$. To the best of our knowledge, no competitive ratio lower bounds have been reported for problems $n \geq 3$.

Some relevant variations to our problem are those in which partial information, e.g. the slope or the distance to the origin, is known regarding the hidden line. All results in this paragraph refer to searching with 1 robot. When both distance and slope are known, the best possible algorithm has competitive ratio 3. When the distance is known, and the line is axis parallel, then the best competitive ratio is $3\sqrt{2}$ [6]. If only the distance is known, [22] gives the best deterministic online algorithm with competitive ratio 6.39. Randomized online algorithms for the same problem were proposed in [20, 21]. The problem in which the slope is known is the traditional cow-path problem with best possible competitive ratio 9 and was studied in [6, 19]. When the line is known to be axis parallel, then [6] gives an upper bound of 13.02, which was improved to 12.5406 [23] and then to 12.5385 [25], the latter shown to be optimal among cyclic-type trajectories. As stated previously, when no information is known the best upper and lower bounds known are 13.81 and (conditionally to cyclic-trajectories) 12.5385, respectively, due to [6] (technical report [17] has a nice exposition of the same upper bound with all mathematical derivations).

Two-dimensional search problems have been considered beyond line searching. Indeed, [21] considered the problem of searching for a circle. In 2010, Langetepe [24] showed that spiral search is optimal for 2 dimensional search by one robot, assuming that all points that are convex combinations of robot's trajectory and the origin are seen/discovered. The same problem with more robots was studied in [18]. Papers [15, 16, 26, 27, 28] consider parallel search on the grid with bounded memory robots. [12] and [29] considered other variations of the problem of searching for a point in the plane, while [30] considered searching for a point within a geometric terrain. Finally, the very recent [31] studied cost/information trade-offs for searching in the plane for a point.

1.2 Problem Definition and Summary of Known and New Results

We begin this section with a formal description of the two-dimensional search problem, first considered in [6].

▶ **Definition 1** (SHORELINE$_n$: Searching for a Shoreline with n Robots).
n unit speed robots start from the origin of the plane. Feasible solutions to the problem are robots' trajectories $\mathcal{F}_n$, such that for every line ℓ of the plane, there exists at least one robot's trajectory intersecting ℓ. The time $T_{\mathcal{F}_n}(\ell)$ by which ℓ is hit for the first time is the search completion time. If $\delta(\ell)$ represents the distance of ℓ to the origin, the objective of SHORELINE$_n$ *is to find trajectories $\mathcal{F}_n$ so as to minimize the search competitive ratio of $\mathcal{F}_n$ defined as*

$$CR(\mathcal{F}_n) := \sup_{\ell} \frac{T_{\mathcal{F}_n}(\ell)}{\delta(\ell)}. \tag{1}$$

The best possible search competitive ratio $\inf_{\mathcal{F}_n} CR(\mathcal{F}_n)$ will be denoted by $\mathcal{S}_n$.

In order to avoid degenerate cases, especially when $n = 1$, the supremum of (1) can be restricted to lines ℓ for which $\delta(\ell) \geq \epsilon$, for some $\epsilon > 0$ that is known to algorithm $\mathcal{F}_n$. Also, for the rest of the paper, and when it is convenient, we will study SHORELINE$_n$ from the perspective of analytic geometry, that is robots will start from the origin of the Cartesian plane, and trajectories will be analytic curves in the plane.

Problem SHORELINE$_n$, and variations of it, have been studied as early as in the late 50's for $n = 1$ and in the 80's for $n \geq 2$. An upper bound to $\mathcal{S}_1$ of 13.81 was reported in [6] and a lower bound of 12.5385 in [25], assuming that the solution trajectory is of spiral-type. The only unconditional lower bound to $\mathcal{S}_1$ is that of 6.3972, see [22], and refers to instances/lines with known distance to the origin, and hence apply to instances with unknown distance as well.

Notably, for the case $n \geq 2$ almost no lower bounds are known and are restricted to instances with known distance to the origin. Indeed, [5] reports that $\mathcal{S}_2 \geq 1.5993$ and the trivial $\mathcal{S}_n \geq 1$, for $n \geq 4$. When it comes to upper bounds, a double-spiral trajectory performed by two robots ensures that $\mathcal{S}_2 \leq 5.2644$, see [5]. For the case $n \geq 3$, [5] proposes the following ray-type algorithm: for $i = 1, \ldots, n$, robot i, searches along the ray with direction $(\cos((i-1)\phi), \sin((i-1)\phi))$, where $\phi = 2\pi/n$. It is an easy exercise to show that this algorithm witnesses that $\mathcal{S}_n \leq \frac{1}{\cos(\pi/n)}$, again for $n \geq 3$.

1.2.1 Organization of the paper

Our main contributions pertain to *new* lower bounds for $\mathcal{S}_n$, when $n \geq 2$, which in particular for $n \geq 4$ are tight. More specifically, we show that $\mathcal{S}_2 \geq 3$ (see Theorem 9 in Section 4), that $\mathcal{S}_3 \geq \sqrt{3}$ (see Theorem 7 in Section 3), and that $\mathcal{S}_n \geq \frac{1}{\cos(\pi/n)}$ (see Theorem 3 in Section 2) for $n \geq 4$. The exposition of the results is in reverse order due to the nature of our arguments. Combined with the known upper bounds discussed above, our results imply the following state-of-the-art regarding problem SHORELINE$_n$, when $n \geq 2$.

▶ **Theorem 2.** *For the best possible competitive ratio $\mathcal{S}_n$ for* SHORELINE$_n$ *we have that:*
$3 \leq \mathcal{S}_2 \leq 5.2644$,
$\sqrt{3} \leq \mathcal{S}_3 \leq 2$,
$\mathcal{S}_n = 1/\cos(\pi/n)$, *for all* $n \geq 4$.

2 Lower Bounds for $n \geq 4$ Robots

This section is devoted to proving the following theorem.

▶ **Theorem 3.** *For all $n \geq 4$, we have $\mathcal{S}_n \geq 1/\cos(\pi/n)$.*

The proof of the theorem above is split in a number of lemmata. First, we show in Lemma 4 that under certain conditions, optimal robots' moves are along straight lines.

▶ **Lemma 4.** *Consider right triangle OMB with $\angle BOM \leq \pi/4$ and $\angle OMB = \pi/2$. Then for every point K in the line segment MB and every point L in the line segment OB we have that $OK + KL \geq OB$, and equality is satisfied only when all points K, B, L coincide.*

Proof. Consider an arbitrary point L in the interior of the line segment OB of right triangle OMB, see also Figure 1. By the law of reflection, among all points K in the interior of segment BM, the one that minimizes $OK + KL$ is the point for which $\angle MKO = \angle LKB$, which angle we also denote by ω. We fix such a point K, and clearly it is enough to show that $OK + KL > OB$. For notational convenience, we introduce abbreviations $t = \angle BOK, y = \angle LKO$ and $\phi = \angle BOM \leq \pi/4$. Note that the arbitrary choice of L uniquely determines angle t.

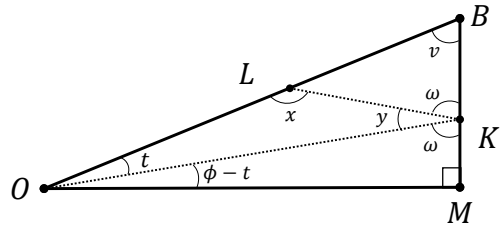


Figure 1 Triangle OMB of the proof of Lemma 4.

First we claim that $OK > OL$. Since these are the sides of triangle KLO, it is enough to prove that for the corresponding opposite angles x, y, respectively, we have $x > y$. Indeed, consider $0 < t < \phi$, and note that $\omega = \pi/2-(\phi-t)$. But then, $y = \pi-2(\pi/2-\phi+t) = 2(\phi-t)$. At the same time, $v = \pi/2 - \phi$, and hence

$$x = v + \omega = \pi/2 - \phi + \omega = \pi - 2\phi + t.$$

Hence, $x > y$ is true if and only if $\pi - 2\phi + t > 2(\phi - t)$ (for every t) or equivalently when $\pi + 3t > 4\phi$. Since $t > 0$, the latter is true for every t as long as $\pi \geq 4\phi$, which is a given premise.

Next we claim that $KL > LB$. Arguing as before, it is enough to establish the same relation for the corresponding opposite angles in triangle BLK, i.e. that $\omega \geq v$. Indeed, since K is in the interior of segment BM and $\angle OMB = \pi/2$, we have $\omega = \angle LKB = \angle MKO > v$.

Summarizing, from the two claims above we have established that $OK > OL$ and $KL > LB$, and hence $OK + KL > OL + LB = OB$, as wanted. ◀

Next, using the lemma above, we show that under certain conditions, there are lines that optimal search trajectories cannot have discovered within certain time bounds.

▶ **Lemma 5.** *Consider trajectories $\mathcal{F}_n$ for problem* Shoreline$_n$*, where robots start from origin O, and fix time $d > 0$. Consider a cone $\mathcal{C}$ of angle 2ϕ, where $0 < \phi \leq \pi/4$, and centered at O. Let also A, B be two points at the two extreme rays of the cone, such that $OA = OB = d + \epsilon$, for some $\epsilon > 0$. If at time d, there is no robot within the cone $\mathcal{C}$, then the line passing through points A, B could not have been intersected by the trajectory of any robot.*

Proof. The proof is by contradiction, so we assume that a robot's trajectory has intersected line ℓ passing through A, B and that the robot is outside cone $\mathcal{C}$. Since we posed the execution of the algorithm at time d, and since $OA = OB > d$, robot's trajectory could not have intersected the extreme rays of cone $\mathcal{C}$ further than points A, B. Since the robot is outside the cone, the trajectory of the robot must have intersected (for the first time) segment AB in some interior point K and then segment OB (or OA) in some interior point L (after hitting line ℓ), see also Figure 2. Also, without loss of generality, K is closer to B than from A. Since robot's trajectory takes place in the Euclidean space, and robot has unit speed, the time for such a trajectory to be realizable is at least $OK + OL$. Consider then the projection M of the origin O onto line segment AB, and observe that $\angle BOM = \phi \leq \pi/4$, since triangle BOA is isosceles. But then, Lemma 4 applies according to which the time that has passed is at least $OK + OL \geq OB = d + \epsilon > d$, a contradiction. ◀

Next we quantify a lower bound to the competitive ratio of search algorithms in which robots exhibit a certain property.

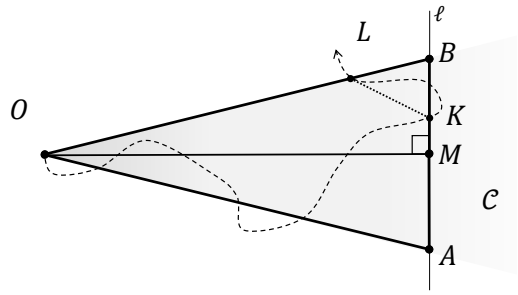


Figure 2 Cone $\mathcal{C}$ of the proof of Lemma 5. Robot's trajectory is depicted as the curved dotted line.

▶ **Lemma 6.** *Consider trajectories $\mathcal{F}_n$ for problem* SHORELINE$_n$*, where robots start from origin O. If there is a cone of angle 2ϕ centered at the origin, where $0 < \phi \leq \pi/4$, within which there is no robot at an arbitrary time (or a robot lies at the origin), then $CR(\mathcal{F}_n) \geq 1/\cos(\phi)$.*

Proof. Consider a time $d > 0$, and a cone centered at the origin of angle 2ϕ, such that no robot lies within the cone. Consider points A, B on the extreme rays of the cone at distance $d + \epsilon$ from the origin, for an arbitrary small $\epsilon > 0$. By Lemma 5, no robot could have discovered the line ℓ passing through points A, B. Since time d has already passed, we conclude that the search completion time satisfies $T_{\mathcal{F}_n}(\ell) > d$. At the same time, triangle OAB is isosceles, and so the distance of ℓ and the origin is $\delta(\ell) = OB \cdot \cos(\phi) = (d+\epsilon)\cos(\phi)$. We conclude that

$$CR(\mathcal{F}_n) \geq \sup_{\epsilon>0} \frac{d}{(d+\epsilon)\cos(\phi)} = \frac{1}{\cos(\phi)}.$$

Finally consider the case that the only robot within the cone lies at the origin. That robot cannot reach the line earlier than $d+(d+\epsilon)\cos(\phi)$, hence the same bound for the competitive ratio holds. ◀

Now we are ready to prove Theorem 3.

Proof of Theorem 3. Fix n, and consider trajectories $\mathcal{F}_n$ for problem SHORELINE$_n$, where robots start from origin O. Let robots move for an arbitrary time $d > 0$. If all robots lie at the origin, then clearly the competitive ratio is unbounded.

Otherwise, consider a cone of arbitrary small angle $2\gamma = o(1/n)$ centered at the origin. We rotate the cone untill at least one robot (note there exists at least one not in the origin) lies strictly within this small cone. Then we cover the plane by concatenating, in an alternate fashion and clockwise to the existing small cone, n many cones centered at the origin of angle $2\pi/n - 2\gamma$, and $n-1$ more cones centered at the origin of angle 2γ.

Note, there are n "small" cones of angle 2γ, one of which strictly contains a robot, and n many "large" cones of angle $2\pi/n - 2\gamma$ and hence one of which, call it $\mathcal{C}$, does not contain any robot, unless a robot is at the origin. But then, Lemma 6 applies with $\phi = 2\pi/n - 2\gamma$, for any γ. That is, for the arbitrary trajectories $\mathcal{F}_n$, and for every $\gamma > 0$ we have that $CR(\mathcal{F}_n) \geq \frac{1}{\cos(\pi/n-\gamma)}$, hence $\mathcal{S}_n \geq \frac{1}{\cos(\pi/n)}$ as wanted. ◀

3 Lower Bound for 3 Robots

In this section we prove the following theorem.

▶ **Theorem 7.** $\mathcal{S}_3 \geq \sqrt{3}$.

Notably, the achieved lower bound does not match the best upper bound known for $\textsc{Shoreline}_3$. In particular, the lower bound arguments of Section 2 fail for $\textsc{Shoreline}_n$, when $n < 4$. Indeed, the crux of the previous argument is that robots should lie at the boundary (extreme rays) of cones, centered at the origin and of angles $2\phi_n := 2\pi/n$. If robots are given, say, time 1 to execute their trajectories, then there are special lines which are $\cos(\phi_n) + \epsilon$ away from the origin, that could not have been visited by any robot, because otherwise the robots would not have enough time to leave from some cones. The crucial necessary condition of the previous statement is that $\phi_n \leq \pi/4$, which of course holds when $n \geq 4$. In the case of $n = 3$, robots can visit these special lines in time less than 1, still leaving the cones of angle $2\pi/3$, hence making the argument invalid. However, the robots would still need a significant amount of time (bounded away from 0) to achieve the same task, hence placing the special lines sufficiently further away would allow the argument to go through.

The paragraph above gives the high level idea of the proof of Theorem 7, and also explains why we presented first, in Section 2, the lower bounds for $n \geq 4$ robots. The next lemma establishes a lower bound for the time that robots need, in $\textsc{Shoreline}_3$, to discover the special lines that were used for the lower bounds to $\mathcal{S}_n$, for $n \geq 4$.

▶ **Lemma 8.** *Consider a cone of angle $2\pi/3$ centered at the origin O, along with two points A, B on its extreme rays at distance 1 from O. Then, a unit speed robot starting from O requires time at least $\sqrt{3}/2$ to visit the line passing through A, B and leave the cone.*

Proof. Consider the projection M of O onto the line ℓ_{AB} passing through A, B. We calculate the shortest trajectory starting from O, visiting an arbitrary point K of line segment MB and leaving the cone from an arbitrary point L of line segment OB, see also Figure 3.

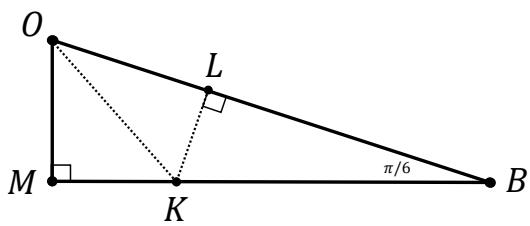


Figure 3 The shortest trajectory, starting from O, visiting line segment MB at point K and leaving the cone from point L from an extreme ray, is depicted with dotted lines.

For convenience, we introduce a coordinate system centered at M, so that $O = (0, 1/2)$ and $B = (\sqrt{3}/2, 0)$ (recall that $OB = 1$). The arbitrary point K on MB is a convex combination of points M, B, and hence has coordinates $K = \lambda(\sqrt{3}/2, 0)$, for some $\lambda \in [0, 1]$. Note that $OK = \sqrt{\frac{3}{4}\lambda^2 + \frac{1}{4}} = \frac{1}{2}\sqrt{3\lambda^2 + 1}$. Given that K is chosen, the shortest path for leaving the cone is clearly the distance $d(K, \ell_{AB})$ between K and ℓ_{OB} passing through O, B. It is easy to see that ℓ_{OB} is described as $y + \sqrt{3}/3x - 1/2 = 0$, hence,

$$d(K, \ell_{OB}) = \frac{\left|\frac{\sqrt{3}}{3}\lambda\frac{\sqrt{3}}{2} - \frac{1}{2}\right|}{\sqrt{1 + \frac{3}{9}}} = \frac{\sqrt{3}}{4}(1 - \lambda).$$

We conclude that the shortest path in order to start from O and leave the cone is

$$\min_{\lambda\in[0,1]} \{OK + d(K, \ell_{OB})\} = \min_{\lambda\in[0,1]} \left\{ \frac{1}{2}\sqrt{3\lambda^2+1} + \frac{\sqrt{3}}{4}(1-\lambda) \right\}$$

We calculate the derivative of the latter function $f(\lambda)$ as $f'(\lambda) = \frac{3\lambda}{2\sqrt{3\lambda^2+1}} - \frac{\sqrt{3}}{4}$, which has a unique root at $\lambda_0 = 1/3$. Then, we calculate $f''(1/3) = \frac{9\sqrt{3}}{16}$, which shows that $\lambda_0 \in [0,1]$ is indeed a minimizer, inducing a trajectory of smallest length $f(1/3) = \sqrt{3}/2$. Since the robot has unit speed, this is also the minimum time needed to reach the cone after visiting line ℓ_{AB}. ◀

We are now ready to prove Theorem 7.

Proof of Theorem 7. Consider trajectories $\mathcal{F}_3$ for problem SHORELINE$_3$, where robots start from origin O. Let robots move for an arbitrary time $d > 0$, and consider 3 cones centered at the origin, each of angle $2\pi/3$, covering the entire plane. We rotate the cones, untill at least one of the robots lies on the extreme ray of two cones. As a result, there is a cone $\mathcal{C}$ such that no robot lies within the interior of the cone.

Now consider points A, B on the extreme rays of $\mathcal{C}$ that are $(2/\sqrt{3}+2\epsilon)d$ away from the origin O, and let the line passing through them be ℓ_{AB}. Note that ℓ_{AB} is exactly $(1/\sqrt{3}+\epsilon)d$ away from the origin.

By Lemma 8, and since no robot lies within $\mathcal{C}$, no robot could have discovered ℓ_{AB} in time d, hence the search completion time is at least d. Overall, that induces competitive ratio for $\mathcal{F}_3$ at least $1/(1/\sqrt{3}+\epsilon)$, for every $\epsilon > 0$. ◀

4 Lower Bound for 2 Robots

In this section we prove a lower bound for searching for a shoreline with 2 robots.

▶ **Theorem 9.** $\mathcal{S}_2 \geq 3$.

In order to prove our theorem, we describe robots' trajectories within time 1. The following function, the boundary of an ellipsoid, will be useful in our calculations

$$q(x, y, \delta, \theta) := 4\left(\cos(\theta)\, x + \sin(\theta)\, y - h_\delta\right)^2 + \left(-\sin(\theta)\, x + \cos(\theta)\, y\right)^2 / b_\delta^2 - 1,$$

where $h_\delta := \delta/2$ and $b_\delta := \sqrt{(1-\delta^2)/2}$.

▶ **Lemma 10.** *Consider an arbitrary algorithm $\mathcal{F}_2$, and let robots execute it for time 1. Then, there exist $\epsilon, \delta \in [0,1]$ and $\theta \in [0,\pi]$ so that no point outside the ellipses $q(x,y,\epsilon,0) \leq 0$, and $q(x,y,\delta,\theta) \leq 0$ has been explored by any robot.*

Proof. Consider an arbitrary algorithm $\mathcal{F}_2$, and let robots execute it for time 1. Suppose that the locations of robots #1, #2 are R_1, R_2 within the unit ball. We claim that the collection of points that each robot could have visited form two ellipses. Indeed, without loss of generality both robots lie in the first two quadrants, i.e. in the non-negative y-axis hyperplane. Also, without loss of generality, the location of R_1, exactly at time 1 of the execution of $\mathcal{F}_2$, equals $(\epsilon, 0)$, for some $\epsilon \in [0,1]$. Recall that robots have speed 1, and they started from the origin $O = (0,0)$. Therefore, all the points P that could have been visited by robot #1 satisfy $OP + PR_1 \leq 1$. In other words, the boundary of the explored domain of that robot is an ellipse with foci points O, R_1 and sum of distances to foci equal to 1. Then, all boundary points $(x,y) \in \mathbb{R}^2$ of the domain that could have been explored by robot #1 satisfy $4(x-h_\epsilon)^2 + y^2/b_\epsilon^2 = 1$.

Similarly, robot #2 is, at the same time, at distance δ from the origin, for some $\delta \in [0,1]$. Arguing as above, the boundary of the explored domain by robot #2 is again an ellipse. So suppose that $R_2 = (\delta\cos(\theta), \delta\sin(\theta))$ for some $\theta \in [0,\pi]$ (that is the line passing through its two foci forms angle θ with the x-axis) Then the boundary of the explored domain by robot #2 is defined as $4(\cos(\theta)x + \sin(\theta)y - h_\delta)^2 + (-\sin(\theta)x + \cos(\theta)y)^2/b_\delta^2 = 1$ (observe that a rotation by angle $-\theta$ gives a formula identical to the one of robot #1). Finally, note that since robot #2 lies in the first two quadrants, we must have $\theta \in [0,\pi]$. ◀

The idea behind the proof of Theorem 9 is that if at a certain time, robots #1, #2 are at points $R_1 = (\epsilon, 0)$ and $R_2 = (\delta\cos(\theta), \delta\sin(\theta))$ (for some $\epsilon, \delta \in [0,1]$ and $\theta \in [0,\pi]$), respectively, then they could not have been in any point past the line $y = -1/2$, see Figure 4 for an example. We are now ready to prove Theorem 9.

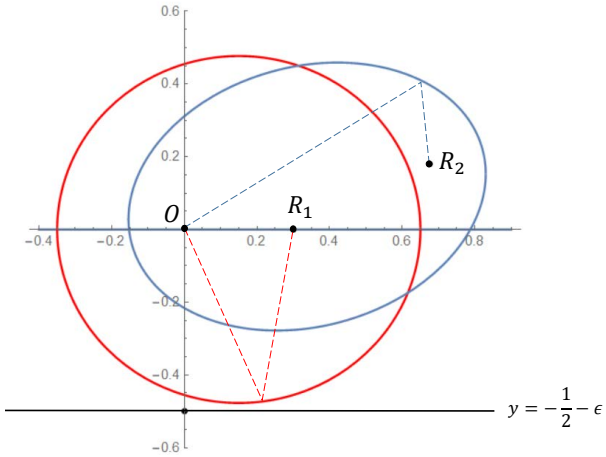


Figure 4 An example of possible robots' placements of an arbitrary search algorithm after time 1. Without loss of generality, robot #1 lies on the positive x-axis, here depicted as point $R_1 = (0.3, 0)$. Similarly, robot #2 lies, without loss of generality, in any of the first two orthants. Here it is depicted as point $R_2 = (0.7\cos(\pi/12), 0.7\sin(\pi/12))$. Both robots have started from the origin, indicated by O. Possible points that robots #1,#2 have visited are depicted by the red and blue ellipses, respectively. Dotted straight trajectories show possible robots' movements from the origin, then to arbitrary points on the boundaries of the ellipses, and then to points R_1, R_2.

Proof of Theorem 9. Consider an arbitrary search algorithm $\mathcal{F}_2$. According to Lemma 10, there are two ellipses that define all points that could have been explored by any of the robots. Our main claim is that neither of the two robots could have hit line $\ell: \ y = -1/2 - \zeta$, where $\zeta > 0$ is arbitrarily small. For that, all we need is to show that none of the equations defining any of the two ellipses have any common point with $y = -1/2 - \zeta$. To that end, we show that equation $q(x, -1/2 - \zeta, \delta, \theta) = 0$ has no real root, when $\delta \in [0,1], \theta \in [0,\pi]$ and $\zeta > 0$ is sufficiently small (that would also imply the same for the first ellipse). Indeed, we compute the discriminant of the degree 2 polynomial $q(x, -1/2 - \zeta, \delta, \theta)$ (in x) which equals

$$-\frac{16\left(\delta^2 + 2\delta(2\zeta+1)\sin(\theta) + 4\zeta(\zeta+1)\right)}{1-\delta^2} \leq -\frac{16\left(\delta^2 + 4\zeta(\zeta+1)\right)}{1-\delta^2}$$

Since $\zeta > 0$ and arbitrarily small, the latter expression is maximized for $\delta = 0$ and becomes $-64\left(\zeta^2 + \zeta\right)$ which is negative for all small enough $\zeta > 0$.

Note that the closest robot to line ℓ is robot #1 (or robot # 2 too, when $\theta = 0, \pi$), and its distance to that line equals $1/2 + \zeta$. Since time 1 has already passed, the search completion time is at least $3/2 + \zeta$. At the same time, the optimal offline solution equals $1/2 + \zeta$. Hence, the competitive ratio of the arbitrary search algorithm $\mathcal{F}_2$ is at least

$$\sup_{\zeta > 0} \frac{3/2 + \zeta}{1/2 + \zeta} = 3. \qquad \blacktriangleleft$$

5 Open Problems

We studied the problem of searching for a shoreline with n robots, and in particular we gave strong lower bounds when $n \geq 2$. Our results are tight when $n \geq 4$, completely resolving these cases. The cases $n = 2, 3$ as well as the case $n = 1$, which is not addressed in this work, remain open. It is plausible that the best algorithms known when $n = 1, 2$ are indeed optimal, even though a proof seems to be particularly challenging. The case of $n = 3$ seems to be the most interesting since the upper bound is provided by the same algorithm as for the cases $n \geq 4$, still our argument that shows optimality for the latter cases fails to be tight when $n = 3$. Finally, a number of variations of the shoreline problem remain open. These include the cases of different robots specs, e.g. speeds, the possibility of faulty robots, different termination criteria, e.g. evacuation or rendezvous instead of search, different measures of efficiency, e.g. average-case worst-case tradeoffs, etc.

References

1. S. Alpern and S. Gal. *The theory of search games and rendezvous*. Springer, 2003.
2. Steve Alpern, Robbert Fokkink, L Gasieniec, Roy Lindelauf, and VS Subrahmanian. *Search theory*. Springer, 2013.
3. Steve Alpern and Shmuel Gal. *The theory of search games and rendezvous*, volume 55. Springer Science & Business Media, 2006.
4. Ricardo Baeza-Yates. Searching: an algorithmic tour. *Encyclopedia of Computer Science and Technology*, 37:331–359, 1997.
5. Ricardo Baeza-Yates and René Schott. Parallel searching in the plane. *Computational Geometry*, 5(3):143–154, 1995.
6. Ricardo A Baeza-Yates, Joseph C Culberson, and Gregory JE Rawlins. Searching with uncertainty. In *Scandinavian Workshop on Algorithm Theory*, pages 176–189. Springer, 1988.
7. Ricardo A Baezayates, Joseph C Culberson, and Gregory JE Rawlins. Searching in the plane. *Information and computation*, 106(2):234–252, 1993.
8. Anatole Beck. On the linear search problem. *Israel Journal of Mathematics*, 2(4):221–228, 1964.
9. Richard Bellman. An optimal search. *Siam Review*, 5(3):274, 1963.
10. Stanley J Benkoski, Michael G Monticino, and James R Weisinger. A survey of the search theory literature. *Naval Research Logistics (NRL)*, 38(4):469–494, 1991.
11. Allan Borodin and Ran El-Yaniv. *Online computation and competitive analysis*. Cambridge University Press, 2005.
12. Sébastien Bouchard, Yoann Dieudonné, Andrzej Pelc, and Franck Petit. Deterministic treasure hunt in the plane with angular hints. In *29th International Symposium on Algorithms and Computation, ISAAC 2018*, volume 123, pages 48–1. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018.
13. J. Czyzowicz, K. Georgiou, and E. Kranakis. Group Search and Evacuation. In P. Flocchini, G. Prencipe, and N. Santoro, editors, *Distributed Computing by Mobile Entities; Current Research in Moving and Computing*, chapter 14, pages 335–370. Springer, 2019.

14 James M Dobbie. A survey of search theory. *Operations Research*, 16(3):525–537, 1968.

15 Y. Emek, T. Langner, J. Uitto, and R. Wattenhofer. Solving the ANTS Problem with Asynchronous Finite State Machines. In *Proceedings of International Colloquium on Automata, Languages, and Programming (ICALP), LNCS 8573*, pages 471–482, 2014.

16 Yuval Emek, Tobias Langner, David Stolz, Jara Uitto, and Roger Wattenhofer. How many ants does it take to find the food? *Theoretical Computer Science*, 608:255–267, 2015.

17 Steven R Finch and Li-Yan Zhu. Searching for a shoreline. *arXiv preprint math/0501123*, 2005.

18 G Matthew Fricke, Joshua P Hecker, Antonio D Griego, Linh T Tran, and Melanie E Moses. A distributed deterministic spiral search algorithm for swarms. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4430–4436. IEEE, 2016.

19 Shmuel Gal. Search games. *Wiley Encyclopedia of Operations Research and Management Science*, 2010.

20 Brian Gluss. An alternative solution to the "lost at sea" problem. *Naval Research Logistics Quarterly*, 8(1):117–122, 1961.

21 Brian Gluss. The minimax path in a search for a circle in a plane. *Naval Research Logistics Quarterly*, 8(4):357–360, 1961.

22 JR Isbell. An optimal search pattern. *Naval Research Logistics Quarterly*, 4(4):357–359, 1957.

23 Artur Jeż and Jakub Łopuszański. On the two-dimensional cow search problem. *Information Processing Letters*, 109(11):543–547, 2009.

24 Elmar Langetepe. On the optimality of spiral search. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 1–12. SIAM, 2010.

25 Elmar Langetepe. Searching for an axis-parallel shoreline. *Theoretical Computer Science*, 447:85–99, 2012.

26 Tobias Langner, Barbara Keller, Jara Uitto, and Roger Wattenhofer. Overcoming Obstacles with Ants. In Emmanuelle Anceaume, Christian Cachin, and Maria Gradinariu Potop-Butucaru, editors, *International Conference on Principles of Distributed Systems (OPODIS)*, volume 46 of *LIPIcs*, pages 9:1–9:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

27 C. Lenzen, N. Lynch, C. Newport, and T. Radeva. Trade-offs Between Selection Complexity and Performance when Searching the Plane Without Communication. In *Proceedings of the Symposium on Principles of Distributed Computing (PODC)*, pages 252–261, 2014.

28 A. López-Ortiz and G. Sweet. Parallel Searching on a Lattice. In *Proceedings of the Canadian Conference on Computational Geometry (CCCG)*, pages 125–128, 2001.

29 Andrzej Pelc. Reaching a Target in the Plane with no Information. *Information Processing Letters*, 140:13–17, 2018.

30 Andrzej Pelc and Ram Narayan Yadav. Information Complexity of Treasure Hunt in Geometric Terrains. *arXiv preprint arXiv:1811.06823*, 2018.

31 Andrzej Pelc and Ram Narayan Yadav. Cost vs. Information Tradeoffs for Treasure Hunt in the Plane. *arXiv preprint arXiv:1902.06090*, 2019.

Gathering on Rings for Myopic Asynchronous Robots With Lights

Sayaka Kamei
Graduate School of Engineering, Hiroshima University, Japan
s10kamei@hiroshima-u.ac.jp

Anissa Lamani
Ecole internationale des sciences du traitement de l'information, Cergy, France
anissa.lamani@eisti.eu

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

Sébastien Tixeuil
Sorbonne University, Paris, France
Sebastien.Tixeuil@lip6.fr

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

Abstract

We investigate gathering algorithms for asynchronous autonomous mobile robots moving in uniform ring-shaped networks. Different from most work using the Look-Compute-Move (LCM) model, we assume that robots have limited visibility and lights. That is, robots can observe nodes only within a certain fixed distance, and emit a color from a set of constant number of colors. We consider gathering algorithms depending on two parameters related to the initial configuration: M_{init}, which denotes the number of nodes between two border nodes, and O_{init}, which denotes the number of nodes hosting robots between two border nodes. In both cases, a border node is a node hosting one or more robots that cannot see other robots on at least one side. Our main contribution is to prove that, if M_{init} or O_{init} is odd, gathering is always feasible with three or four colors. The proposed algorithms do not require additional assumptions, such as knowledge of the number of robots, multiplicity detection capabilities, or the assumption of towerless initial configurations. These results demonstrate the power of lights to achieve gathering of robots with limited visibility.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases LCM robot system, ASYNC schedulers, myopic, luminous, ring networks

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.27

Related Version A full version of the paper is available at [13], `http://arxiv.org/abs/1911.04757`.

Funding Supported in part by JST SICORP Grant Number JPMJSC1806, Japan.
Sayaka Kamei: Supported in part by JSPS KAKENHI No. 19K11828.
Fukuhito Ooshita: Supported in part by JSPS KAKENHI No. 18K11167.
Sébastien Tixeuil: Supported in part by Hiroshima University and by project ESTATE (Ref. ANR-16-CE25-0009-03).
Koichi Wada: Supported in part by JSPS KAKENHI No. 17K00019.

© Sayaka Kamei, Anissa Lamani, Fukuhito Ooshita, Sébastien Tixeuil, and Koichi Wada;
licensed under Creative Commons License CC-BY
23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller; Article No. 27; pp. 27:1–27:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

A lot of research about autonomous mobile robots coordination has been conducted by the distributed computing community. The common goal of these research is to clarify the minimum capabilities for robots to achieve a given task. Hence, most work adopts weak assumptions such as: robots are identical (*i.e.*, robots execute the same algorithm and cannot be distinguished), oblivious (*i.e.*, robots have no memory to record past actions), and silent (*i.e.*, robots cannot send messages to other robots). In addition, to model the behavior of robots, most work uses the Look-Compute-Move (LCM) model introduced by Suzuki and Yamashita [18]. In the LCM model, each robot repeats executing cycles of Look, Compute and Move phases. During the Look phase, the robot takes a snapshot to observe the positions of other robots. According to this snapshot, the robot computes the next movement during the Compute phase. If the robot decides to move, it moves to the target position during the Move phase. By using the LCM model, it is possible to clarify problem solvability both continuous environments (*i.e.*, two- or three-dimensional Euclidean space) and discrete environments (*i.e.*, graph networks). State-of-the-art surveys are given in the recent book by Flocchini *et al.* [8].

In this paper, we focus on gathering in graph networks. The goal of gathering is to make all robots gather at a non-predetermined single node. Since gathering is a fundamental task of mobile robot systems and a benchmark application, numerous algorithms have been proposed for various graph network topologies. In particular, many papers focus on ring-shaped networks because symmetry breaking becomes a core difficulty, and any such solution is likely to adapt well on other topologies, as it is possible to make virtual rings over arbitrary networks and hence use ring algorithms in such networks[14, 11, 5, 3, 4].

Klasing *et al.* [14] proposed gathering algorithms for rings with *global-weak* multiplicity detection. Global-weak multiplicity detection enables a robot to detect whether the number of robots on each node is one, or more than one. However, the exact number of robots on a given node remains unknown if there is more than one robot on the node. Izumi *et al.* [11] provided a gathering algorithm for rings with *local-weak* multiplicity detection under the assumption that the initial configurations are non-symmetric and non-periodic, and that the number of robots is less than half the number of nodes. Local-weak multiplicity detection enables a robot to detect whether the number of robots on its *current* node is one, or more than one. D'Angelo *et al.* [5, 3] proposed unified ring gathering algorithms for most of the solvable initial configurations, using global-weak multiplicity detection [5], or local-weak multiplicity detection [3]. Finally, Klasing *et al.* [4] proposed gathering algorithms for grids and trees. All aforementioned work assumes *unlimited visibility*, that is, each robot can take a snapshot of the whole network graph with all occupied positions.

The unlimited visibility assumption somewhat contradicts the principle of weak mobile robots, hence several recent studies focus on *myopic* robots [9, 10, 12, 7]. A myopic robot has limited visibility, that is, it can take a snapshot of nodes (with occupying robots) only within a certain fixed distance ϕ. Not surprisingly, many problems become impossible to solve in this setting, and several strong assumptions have to be made to enable possibility results. Datta *et al.* [7] study the problem of ring exploration with different values for ϕ. Guilbault *et al.* [9] study gathering in bipartite graphs with the global-weak multiplicity detection (limited to distance ϕ) in case of $\phi = 1$, and prove that gathering is feasible only when robots form a star in the initial configuration. They also study the case of infinite lines with $\phi > 1$ [10], and prove that no universal algorithm exists in this case. In the case of rings, since a ring with even nodes is also a bipartite graph, gathering is feasible only

Table 1 Summary of gathering algorithms for myopic robots in rings. All algorithms assume that the visibility graph is connected, and that there exist two border nodes. R denotes the number of robots and H_{center} denotes the size of the center hole in the initial configuration. The global (resp., local) strong multiplicity detection enables a robot to observe the exact number of robots on every node (resp., on its current node).

	ϕ	#colors	Multiplicity detection	Assumptions
[12]	≥ 1	1	local strong	R is odd R is known towerless initial configuration
[12]	≥ 3	1	global strong	R is even R is known H_{center} is odd $H_{center} \leq \phi - 2$ towerless initial configuration
Algorithm 1	≥ 1	3	none	M_{init} is odd
Algorithm 2	≥ 2	4	none	O_{init} is odd

when three robots occupy three successive nodes. For this reason, Kamei *et al.* [12] give gathering algorithms for rings with $\phi \geq 1$ by using strong assumptions, such as knowledge of the number of robots, and strong multiplicity detection, which enables a robot to obtain the exact number of robots on a particular node. Overall, limited visibility severely hinders the possibility of gathering oblivious robots on rings.

On the other hand, completely oblivious robots (that can not remember past actions) may be too weak of a hypothesis with respect to a possible implementation on real devices, where persistent memory is widely available. Recently, enabling the possibility that robots maintain a non-volatile visible light [6] has attracted a lot of attention to improve the task solvability. A robot endowed with such a light is called a *luminous* robot. Each luminous robot can emit a light to other robots whose color is chosen among a set of colors whose size is constant. The light color is non-volatile, and so it can be used as a constant-space memory. Viglietta [19] gives a complete characterization of the rendezvous problem (that is, the gathering of two robots) on a plane using two visible colored lights assuming unlimited visibility robots. Das *et al.* [6] prove that unlimited visibility robots on two-dimensional space with a five-color light have the same computational power in the asynchronous and semi-synchronous models. Di Luna *et al.* [15] discuss how lights can be used to solve some classical distributed problems such as rendezvous and forming a sequence of patterns. The robots they assume have unlimited visibility, but they also discuss the case where robots visibility is limited by the presence of obstruction. Hence, luminous robots seem to dramatically improve the possibility to solve tasks in the LCM model.

As a result of the above observations, it becomes interesting to study the interplay between the myopic and luminous properties for LCM robots: can lights improve task solvability of myopic robots. To our knowledge, only three papers [17, 16, 2] consider this combination. Ooshita *et al.* [17] and Nagahama *et al.* [16] demonstrate that for the task of ring exploration, even a two-color light significantly improves task solvability. Bramas *et al.* [2] give exploration algorithms for myopic robots in infinite grids by using a constant-color light. To this day, the characterization of gathering feasibility for myopic luminous robots (aside from the trivial case where a single color is available) is unknown.

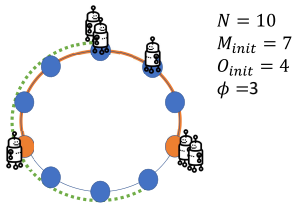


Figure 1 Example of an initial configuration where $M_{init} = 7$ and $O_{init} = 4$. Border nodes are represented in orange color. The robot on the left border node can sense the range represented by green dots.

Our contributions. We clarify the solvability of gathering for myopic luminous robots in rings. We consider the asynchronous (ASYNC) model, which is the most general timing assumption. As in a previous work [12], we focus on initial configurations such that the visibility graph[1] is connected and there exist two border nodes[2] (see Figure 1 for an example). Both assumptions are necessary for the class of *cautious* gathering algorithms (see Lemmas 24 and 25). A cautious gathering protocol never expands the span of a given visibility graph. In addition, we assume that all robots have the same color in initial configurations. We partition initial configurations using two parameters M_{init} and O_{init}; M_{init} is defined as the number of nodes between two border nodes, and O_{init} is defined as the number of nodes occupied by some robots (also see Figure 1). We can easily observe that, if both M_{init} and O_{init} are even, there exist (so-called *edge-symmetric*) initial configurations such that no algorithm achieves gathering (Corollary 22). Hence, we consider the case that M_{init} or O_{init} is odd.

On the positive side, our main contribution is to prove that, if M_{init} or O_{init} is odd, gathering is always feasible by using a constant number of colors without additional assumptions (so, no multiplicity detection is necessary) for any positive visible distance ϕ. First, for the case that M_{init} is odd and $\phi \geq 1$ holds, we give a gathering algorithm that uses three colors. Second, for the case that O_{init} is odd and $\phi \geq 2$, we give a gathering algorithm that uses four colors. Note that we assume $\phi \geq 2$ in the second algorithm because, if $\phi = 1$ holds, then $O_{init} = M_{init}$ also holds from the assumption of connected visibility graphs, so the first algorithm can be used in this case. We compare the current work with that of Kamei *et al.* [12] in Table 1. Overall, lights with a constant number of colors permit to remove most of the previously considered assumptions. For example, our algorithms do not require any multiplicity detection (that is, robots do not distinguish whether the number of robots with the same color on a single node is one or more than one). Furthermore, our algorithms solve gathering even if initial configurations include tower nodes (a tower node is a node that hosts multiple robots). These results demonstrate the power of lights to achieve mobile robot gathering with limited visibility.

Because of space limitations, we give proofs of lemmas and theorems in [13].

2 Model

We consider anonymous, disoriented and undirected rings $\mathcal{G}$ of $N(\geq 3)$ nodes $u_0, u_1, \ldots, u_{N-1}$ such that u_i is connected to both $u_{((i-1) \mod N)}$ and $u_{((i+1) \mod N)}$. On this ring, R autonomous robots collaborate to gather at one of the N nodes of the ring, not known beforehand, and remain there indefinitely.

[1] A visibility graph is defined as $\mathcal{G}_V = (\mathcal{R}, \mathcal{E}_V)$ where $\mathcal{R}$ is a set of all robots, and $\mathcal{E}_V$ is a set of pairs of robots that can observe each other.

[2] Node v is a border node if robots on v can observe other robots only in one direction.

The *distance* between two *nodes* u and v on a ring is the number of edges in a shortest path connecting them. The *distance* between two *robots* r_1 and r_2 is the distance between two nodes occupied by r_1 and r_2, respectively. Two robots or two nodes are *neighbors* if the distance between them is one.

Robots are identical, *i.e.*, they execute the same program and use no localized parameter such as an identifier or a particular orientation. Also, they are oblivious, *i.e.*, they cannot remember their past observations or actions. We assume that robots do *not* know N, the size of the ring, and R, the number of robots.

Each robot r_i maintains a variable L_i, called *light*, which spans a finite set of states called *colors*. A light is *persistent* from one computational cycle to the next: the color is not automatically reset at the end of the cycle. Let K denote the number of available light colors. Let $L_i(t)$ be the light color of r_i at time t. We assume the *full light* model: each robot r_i can see the light of other robots, but also its own light. Robots are unable to communicate with each other explicitly (*e.g.*, by sending messages), however, they can observe their environment, including the positions and colors of the other robots. We assume that besides colors, robots do *not* have multiplicity detection capability: if there are multiple robots $r_1, r_2, \ldots r_k$ in a node u, an observing robot r can detect only colors, so r can detect there are multiple robots at u if and only if at least two robots among $r_1, r_2, \ldots r_k$ have different colors. So, a robot r observing a single color at node u cannot know how many robots are located in u.

Based on the sensing result, a robot r may decide to move or to stay idle. At each time instant $t_i (1 \leq i)$, robots occupy nodes of the ring, their positions and colors form a *configuration* $C(t_i)$ of the system at time t_i. When $C(t_i)$ reaches $C(t_{i+1})$ by executing some phases between t_i and t_{i+1}, it is denoted as $C(t_i) \longrightarrow C(t_{i+1})$. The reflexive and transitive closure is denoted as $\longrightarrow^*$.

We assume that robots have limited visibility: an observing robot r at node u can only sense the robots that occupy nodes within a certain distance, denoted by ϕ ($\phi \geq 0$), from u. As robots are identical, they share the same ϕ.

Let $\mathcal{X}_i(t)$ be the set of colors of robots located in node u_i at time t. If a robot r_j located at u_i at t, the sensor of r_j outputs a sequence, $\mathcal{V}_j$, of $2\phi + 1$ set of colors:

$$\mathcal{X}_{i-\phi}(t), \ldots, \mathcal{X}_{i-1}(t), (\mathcal{X}_i(t)), \mathcal{X}_{i+1}(t), \ldots, \mathcal{X}_{i+\phi}(t).$$

This sequence $\mathcal{V}_j$ is the *view* of r_j. If the sequence $\mathcal{X}_{i+1}, \ldots, \mathcal{X}_{i+\phi}$ is equal to the sequence $\mathcal{X}_{i-1}, \ldots, \mathcal{X}_{i-\phi}$, then the view $\mathcal{V}_j$ of r_j is *symmetric*. Otherwise, it is *asymmetric*. In $\mathcal{V}_j$, a node u_k is *occupied* at instant t whenever $|\mathcal{X}_k(t)| > 0$. Conversely, if u_k is not occupied by any robot at t, then $\mathcal{X}_k(t) = \emptyset$ holds, and u_k is *empty* at t.

If there exists a node u_i such that $|\mathcal{X}_i(t)| = 1$ holds, u_i is *singly-colored*. Note that $|\mathcal{X}_i(t)|$ denotes the number of colors at node u_i, thus even if u_i is singly-colored, it may be occupied by multiple robots (sharing the same color). Now, if a node u_i is such that $|\mathcal{X}_i(t)| > 1$ holds, u_i is *multiply-colored*. As each robot has a single color, a multiply-colored node always hosts more than one robot.

In the case of a robot r_j located at a singly-colored node u_i, r_j's view $\mathcal{V}_j$ contains an $\mathcal{X}_i(t)$ that can be written as $[L_j]$. Then, if the left node of u_i contains one or more robots with color L_k, and the right node of u_i contains one or more robots with color L_l, while u_i only hosts r_j, then $\mathcal{V}_j$ can be written as $L_k[L_j]L_l$. Now, if robot r_j at node u_i occupies a multiply-colored position (with two other robots r_k and r_l having distinct colors), then

$|\mathcal{X}_i(t)| = 3$, and we can write $\mathcal{X}_i(t)$ in $\mathcal{V}_j$ as $\begin{bmatrix} L_k \\ L_l \\ [L_j] \end{bmatrix}$. When the view does not consist of a single observed node, we use brackets to distinguish the current position of the observing robot in the view and the inner bracket to explicitly state the observing robot's color.

Our algorithms are driven by observations made on the current view of a robot, so many instances of the algorithms we use *view predicates*: a Boolean function based on the current view of the robot. The predicate L_j matches any set of colors that includes color L_j, while predicate (L_j, L_k) matches any set of colors that contains L_j, L_k, or both. Now the predicate $\begin{pmatrix} L_1 \\ L_2 \end{pmatrix}$ matches any set that contains *both* L_1 and L_2. Some of our algorithm rules expect that a node is singly-colored, *e.g.* with color L_k, in that case, the corresponding predicate is denoted by $L_k!$. To express predicates in a less explicit way, we use character '?' to represent any non-empty set of colors, so a set of colors $\mathcal{X}_i \neq \emptyset$ satisfies predicate '?'. The $\neg$ operator is used to negate a particular predicate P (so, $\neg P$ returns *false* whenever P returns *true* and vice versa). Also, the superscript notation P^y represents a sequence of y consecutive sets of colors, each satisfying predicate P. Observe that $y \leq \phi$. In a given configuration, if the view of a robot r_j at node u_i satisfies predicate $\emptyset^\phi[?]$ or predicate $[?]\emptyset^\phi$, then r_j is a *border robot* and u_i a *border node*. Sometimes, we require a particular color L_1 to be present at some position u_i and a particular color L_2 *not* to be present at some position u_i. For the above case, the corresponding predicate would be: $\begin{pmatrix} L_1 \\ \neg L_2 \end{pmatrix}$.

In this paper, we aim at maintaining the property that at most two border nodes exist at any time. On the ring $\mathcal{G}$, let H_{max} be the size of the maximum hole (*i.e.*, the maximum sequence of empty nodes). Note that by the assumptions, at instant $t = 0$ (*i.e.*, in the initial configuration), $H_{max} > \phi$ holds. Let $\mathcal{V}'$ be the subset of nodes on a path between two border nodes u and v, such that all robots are hosted by nodes in $\mathcal{V}'$. Also, let $\mathcal{G}'$ be the subgraph of $\mathcal{G}$ induced by $\mathcal{V}'$. Note that, $\mathcal{G}'$ does not include the hole with the size H_{max}. At instant $t = 0$, let H_{init} be the maximum distance between occupied nodes in $\mathcal{G}'$, M_{init} be the number of nodes in $\mathcal{G}'$, and $O_{init}(\leq M_{init})$ be the number of occupied nodes in $\mathcal{G}'$. We assume that $\phi \geq H_{init} \geq 1$ holds. Note that, H_{init} is the size of the second maximum hole in the ring because there are two border nodes. As previously stated, no robot is aware of H_{init}, M_{init} and O_{init}. In $\mathcal{G}'$, let D denote the distance between the two border nodes. Note that, at $t = 0$, $D = M_{init} - 1$ holds.

Each robot r executes Look-Compute-Move cycles infinitely many times: (*i*) first, r takes a snapshot of the environment and obtains an ego-centered view of the current configuration (Look phase), (*ii*) according to its view, r decides to move or to stay idle and possibly changes its light color (Compute phase), (*iii*) if r decided to move, it moves to one of its neighbor nodes depending on the choice made in the Compute phase (Move phase). At each time instant t, a subset of robots is activated by an entity known as the scheduler. This scheduler is assumed to be fair, *i.e.*, all robots are activated infinitely many times in any infinite execution. In this paper, we consider the most general asynchronous model: the time between Look, Compute, and Move phases is finite but unbounded. We assume however that the move operation is atomic, that is, when a robot takes a snapshot, it sees robots colors on nodes and not on edges. Since the scheduler is allowed to interleave the different phases between robots, some robots may decide to move according to a view that is different from the current configuration. Indeed, during the compute phase, other robots may move. Both the view and the robot are in this case said to be *outdated*.

In this paper, each rule in the proposed algorithms is presented in the similar notation as in [17]: $< Label > : < Guard > :: < Statement >$. The guard is a predicate on the view $\mathcal{V}_j = \mathcal{X}_{i-\phi}, \ldots, \mathcal{X}_{i-1}, (\mathcal{X}_i), \mathcal{X}_{i+1}, \ldots, \mathcal{X}_{i+\phi}$ obtained by robot r_j at node u_i during the Look phase. If the predicate evaluates to *true*, r_j is *enabled*, otherwise, r_j is *disabled*. In the first case, the corresponding rule $< Label >$ is also said to be *enabled*. If a robot r_j is enabled, r_j may change its color and then move based on the corresponding statement during its subsequent Compute and Move phases. The statement is a pair of (*New color*, *Movement*). Movement can be (i) $\rightarrow$, meaning that r_j moves towards node u_{i+1}, (ii) $\leftarrow$, meaning that r_j moves towards node u_{i-1}, and (iii) $\perp$, meaning that r_j does not move. For simplicity, when r_j does not move (resp. r_j does not change its color), we omit *Movement* (resp. *New color*) in the statement. The label $< Label >$ is denoted as R followed by a non-negative integer (*i.e.*, R0, R1, etc.) where a smaller label indicates higher priority.

3 Algorithms

In this section, we propose two algorithms for myopic luminous robots. One is for the case that M_{init} is odd, uses three colors ($K = 3$). The other is for the case that M_{init} is even and O_{init} is odd, uses four colors ($K = 4$). We assume that the initial configurations satisfy the following conditions:

- All robots have the same color White,
- Each occupied node can have multiple robots, and
- $\phi \geq H_{init} \geq 1$ holds.

3.1 Algorithm for the case M_{init} is odd

The strategy of our algorithm is as follows: The robots on two border nodes keep their lights Red or Blue, then the algorithm can recognize that they are originally on border nodes. When robots on a border node move toward the center node, they change the color of their light to Blue or Red alternately regardless of the neighboring nodes being occupied, where initially robots become Red colors. To keep the connected visibility graph, when a border node becomes singly-colored, the border robot changes its light to Blue or Red according to the distance from the original border node and moves toward the center node and the neighboring non-border robot becomes a border robot. Eventually, two border nodes become neighboring. Then, one has Blue robots and the other has Red robots because M_{init} is odd. At the last moment, Red robots join Blue robots to achieve the gathering.

The formal description of the algorithm is in Algorithm 1. The rules of our algorithm are as follows:

- R0: If the gathering is achieved, a robot does nothing[3].
- R1: A border White robot on a singly-colored border node changes its light to Red.
- R2a and R2b: A border Red robot on a singly-colored border node changes its light to Blue and moves toward an occupied node.
- R3a and R3b: A border Blue robot on a singly-colored border node changes its light to Red and moves toward an occupied node.

[3] Note, this algorithm and the next one cannot terminate because gathering configurations are not terminating ones due to robots with outdated views even if this rule is executed. Because this rule has higher priority, if it is enabled, robots do not need to check other guards.

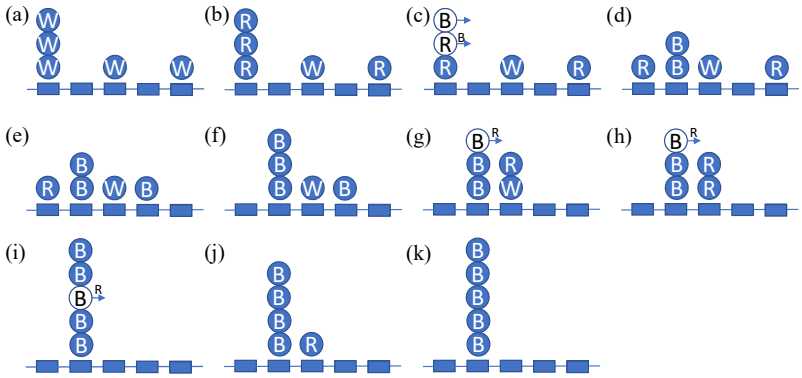


Figure 2 An execution example of Algorithm 1 with $\phi = 2$.

- R4a and R4b: When White robots become border robots, they change their color to the same color as the border Red or Blue robots.
- R5: If two border nodes are neighboring, a border Red robot on a singly-colored border node moves to the neighboring singly-colored node with Blue robots.

Figure 2 illustrates an execution example of Algorithm 1. This figure assumes $\phi = 2$. Figure 2(a) shows an initial configuration. First, border White robots change their lights to Red by R1 (Fig. 2(b)). Next, left border Red robots move by R2a. Since we consider the ASYNC model, some robots may become outdated. In Fig. 2(c), the top robot has changed its light but not yet moved, and the middle robot has looked but not yet changed its light. The outdated robots move in Fig. 2(d), and then the right border Red robot also moves in Fig. 2(e) by R2a. Here, one left border Red robot has not yet moved. However, since it still observes a White robot, it can move by R2a (Fig. 2(f)). Then, border Blue robots can move by R3b. In Fig. 2(g), one left border robot becomes outdated and the right border robot completes the movement. After the right border White robot changes its light to Red by R4a (Fig. 2(h)), the right border Red robots move by R5 (Fig. 2(i)). Note that, all robots stay at a single node but one of them is outdated. Hence the robot moves after that (Fig. 2(j)), but it can go back to the gathering node by R5 (Fig. 2(k)). Now robots have achieved gathering.

In the case of $\phi = 1$, each robot can view only its neighboring nodes. In this case, we should add some assumptions as follows:

- R2a and R3a are always disabled.
- R2b and R3b are enabled when there are White robots in the neighboring node.
- R5 is enabled when the neighboring node is singly-colored with Blue robots.

In that case, because of the connectivity of the visibility graph, there is no empty node in $\mathcal{G}'$. That is, until gathering is achieved, there is at least one White robot on the neighboring node. Thus, such assumption is natural and we can prove the correctness in the same way as other cases by deleting R2a and R3a (the case such that there is no White robot on the neighboring node but gathering is not achieved).

We now discuss the correctness of our algorithm. Because there is no rule such that non-border robot r_i can execute, we can derive the following lemma.

▶ **Lemma 1.** *When a robot r_i looks, if it is a non-border robot, it cannot execute any action.*

To discuss the correctness, we consider the time instants such that the distance D between two border nodes has just reduced at least one. The duration between them is called *mega-cycle*. Note that, D is reduced at most two by the algorithm during a mega-cycle.

Algorithm 1 Algorithm for the case that M_{init} is odd.

Colors
W (White), R (Red), B (Blue)

Rules

/* Do nothing after gathering. */
R0: $\emptyset^{\phi}[?]\emptyset^{\phi} :: \perp$

/* Start by the initial border robot. */
R1: $\emptyset^{\phi}[W!](\neg\emptyset^{\phi}) :: R$

/* Border robots on singly-colored nodes change their color alternately and move. */
R2a: $\emptyset^{\phi}[R!](\emptyset, B!)(\neg(\emptyset^{\phi-1})) :: B, \rightarrow$

R2b: $\emptyset^{\phi}[R!](W)(?^{\phi-1}) :: B, \rightarrow$

R3a: $\emptyset^{\phi}[B!](\emptyset, R!)(\neg(\emptyset^{\phi-1})) :: R, \rightarrow$

R3b: $\emptyset^{\phi}[B!](W)(?^{\phi-1}) :: R, \rightarrow$

/* When White robots become a border robot, they change their color to the same color as the border robot. */
R4a: $\emptyset^{\phi}\begin{bmatrix} R \\ [W] \end{bmatrix}(\neg\emptyset^{\phi}) :: R$

R4b: $\emptyset^{\phi}\begin{bmatrix} B \\ [W] \end{bmatrix}(\neg\emptyset^{\phi}) :: B$

/* When two border nodes are neighboring, robots gather to a node with the Blue border robots. */
R5: $\emptyset^{\phi}[R!](B!)(\emptyset^{\phi-1}) :: B, \rightarrow$

Let $t_0, t_1, \ldots t_i, \ldots$ be starting times of mega-cycles, where t_0 is the starting time of the algorithm and for each i (≥ 1), D is reduced at least one from t_{i-1} at time t_i. Letting $C(t_i)$ be the configuration at time t_i, the transition of configurations from t_i to t_{i+1} is denoted as $C(t_i) \xrightarrow{MC} C(t_{i+1})$.

Figure 3 shows a transition diagram of configurations for each mega-cycle (We prove all transitions later). The small blue box represents a node, and the circle represents a set of robots (One single circle may represent a set of robots with the same color). The doubly (resp. singly) lined arrows represent that D decreases by 2 (resp. 1). The letter in each circle represents the color of the lights. The circles in parentheses represent that they are optional. The circles in brackets represent that one of them should exist. In Conf 1-3, there is no Red or Blue robot with an outdated view (*i.e.*, all robots are after Move phases before they look). In Conf 4-7, left side border node represents that there is no Red or Blue robot with an outdated view. Right side border robots in Conf 4-7 represent that they are still working on their movements, *i.e.*, there may be robots with outdated views. The second node from the right can be empty, and there can be White robots on the node. In the right side border node, the white circles represent that the robots can be on the node, but with the outdated view. Actually, there may be White robots which do not change their color to the same as the border color, Red or Blue, yet. We omit such White robots changing to border color for simplicity, *i.e.*, they may be included in any set of White robots in border nodes in this figure[4]. For example, consider an example in Fig. 2. The initial configuration in Fig. 2(a) is represented by Init. The next mega-cycle starts in Fig. 2(e) and this configuration is represented by Conf 4. Note that mapping from left and right border nodes in Conf 1-7 to two border nodes in configurations may change during an execution. The next mega-cycle starts in Fig. 2(f) and this configuration is represented by Conf 1.

Because the border node becomes singly-colored by R4, we can derive the following lemma.

[4] However, it is considered in the proof.

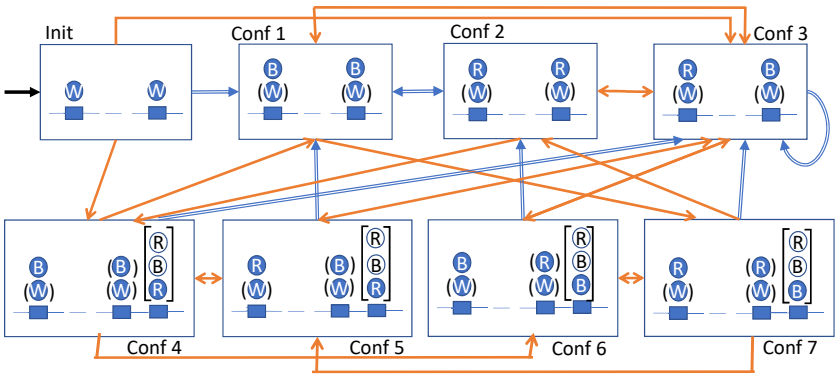


Figure 3 Transition Diagram for Algorithm 1 while $D > 2$.

▶ **Lemma 2.** *Assume that a border node contains White and γ robots, where $\gamma \in \{Red, Blue\}$ at time t. Then there exists a time $t'(> t)$ such that the border node becomes singly-colored one with γ robots. In the case that the border node contains only White robots, the border node becomes Red singly-colored.*

Because border robots on a singly-colored node eventually move to a neighboring node by R2 or R3, the following lemma holds.

▶ **Lemma 3.** *Assume that a border node u is singly-colored with Red (resp. Blue) at time t, then if there is no robot in u at time $t'(> t)$, all robots in u change their color to Blue (resp. Red), move to the neighboring node of u, and the distance between the border nodes is reduced by at least one at t'.*

We showed that the transition diagram in Figure 3 is correct while $D > 2$. In each mega-cycle, D decreases by at least one, and border robots change their color Red and Blue alternately every hop.

▶ **Lemma 4.** *From the initial configuration, D decreases monotonically and eventually becomes 2.*

▶ **Lemma 5.** *Let h be the distance from the original border node to a node u_h in $\mathcal{G}'$. If h is odd (resp. even), a Blue (resp. Red) border robot comes into u_h.*

In the following part, we consider the execution after D becomes 2.

Let Conf BW-MR be the configuration with $D = 1$ such that there are Blue robots and White robots without outdated views in a border node and there are Red robots and Blue robots with outdated views and Red robots without outdated views in the other border node, where Blue robots with outdated views will move to the other border node and Red robots with outdated views will change their color to Blue and move to the other border node (Figure 4a).

Let Conf RW-MB be the configuration with $D = 1$ such that there are Red robots and White robots without outdated views in a border node and there are Red robots and Blue robots with outdated views and Blue robots without outdated views in the other border node, where Red robots with outdated views will move to the other border node and Blue robots with outdated views will change their color to Red and move to the other border node (Figure 4b).

▶ **Lemma 6.** *After D becomes 2, the configuration becomes Conf BW-MR or Conf RW-MB.*

(a) Conf BW-MR **(b)** Conf RW-MB

Figure 4 The configurations Conf BW-MR and Conf RW-MB with $D = 1$.

To show that the gathering is achieved, by Lemmas 4 and 6, we consider the gathering only from Conf BW-MR and Conf RW-MB respectively.

▶ **Lemma 7.** *From Conf BW-MR, the gathering is achieved.*

▶ **Lemma 8.** *From Conf RW-MB, the gathering is achieved.*

By Lemmas 7 and 8, we can derive the following theorem.

▶ **Theorem 9.** *Gathering is solvable in full-light of 3 colors when M_{init} is odd.*

3.2 Algorithm for the case M_{init} is even and O_{init} is odd

In this case, we can assume $\phi > 1$ because the visibility graph is connected. The strategy of our algorithm is similar to Algorithm 1. Initially, all robots are White, and robots on two border nodes become Red in their first activation. The two border robots keep their lights Red or Blue, then the algorithm can recognize that they are originally border robots. When non-border White robots become border robots, they change their color to Red (resp., Blue) if borders that join the node have Red (resp., Blue). To keep the connected visibility graph, when a border node becomes singly-colored, the border robot moves toward the center node. At that time, if there exists a White robot in the directed neighboring node, the border robot changes its color. Otherwise, it just moves without changing its color. Eventually, two border nodes become neighboring. In this algorithm, when two border nodes are neighboring, an additional color Purple is used to decide the gathering point.

The rules of our algorithm are as follows:

- R0: If the gathering is achieved, a robot does nothing.
- R1: A border White robot on a singly-colored border node changes its light to Red.
- R2: A border Red robot on a singly-colored border node moves toward an occupied node without changing its color when there is no White robot on the neighboring node (R2a-1, R2a-2). A border Red robot moves toward an occupied node and changes its light to Blue only when there is at least one White robot on the neighboring node (R2b).
- R3: A border Blue robot on a singly-colored border node moves toward an occupied node without changing its color when there is no White robot on the neighboring node (R3a-1, R3a-2). A border Blue robot moves toward an occupied node and changes its light to Red only when there is at least one White robot on the neighboring node (R3b).
- R4: When White robots become border robots, they change their color to the same color as the border Red or Blue robots.
- R5: If two border nodes are neighboring, every robot moves to the neighboring node with Purple robots (R5a). A border Blue robot on a singly-colored border node changes its light to Purple when there are only Red robots or Red and Blue robots on the neighboring node (R5b-1, R5b-2). A border Blue robot changes its light to Purple when there is Red robot on the same node and the neighboring node is a singly-colored node with Red robots (R5b-3).

Algorithm 2 Algorithm for the case that M_{init} is even and O_{init} is odd.

Colors

W (White), R (Red), B (Blue), P (Purple)

Rules

/* Do nothing after gathering. */
R0: $\emptyset^{\phi}[?]\emptyset^{\phi} :: \bot$

/* Start by the initial border robots. */
R1: $\emptyset^{\phi}[W!](\neg\emptyset^{\phi}) :: R$

/* Border robots on singly-colored nodes move inwards. */
R2a-1: $\emptyset^{\phi}[R!](\emptyset)(\neg(\emptyset^{\phi-1})) :: \rightarrow$

R2a-2: $\emptyset^{\phi}[R!]\begin{pmatrix}\neg W\\ R\end{pmatrix}(\neg(\emptyset^{\phi-1})) :: \rightarrow$

R2b: $\emptyset^{\phi}[R!](W)(?^{\phi-1}) :: B, \rightarrow$

R3a-1: $\emptyset^{\phi}[B!](\emptyset)(\neg(\emptyset^{\phi-1})) :: \rightarrow$

R3a-2: $\emptyset^{\phi}[B!]\begin{pmatrix}\neg W\\ B\end{pmatrix}(\neg(\emptyset^{\phi-1})) :: \rightarrow$

R3b: $\emptyset^{\phi}[B!](W)(?^{\phi-1}) :: R, \rightarrow$

/* When White robots become border robots, they change their color to the same color as the border robots. */
R4a: $\emptyset^{\phi}\begin{bmatrix}R\\ [W]\end{bmatrix}(\neg\emptyset^{\phi}) :: R$

R4b: $\emptyset^{\phi}\begin{bmatrix}B\\ [W]\end{bmatrix}(\neg\emptyset^{\phi}) :: B$

/* When two border nodes are neighboring, they gather to the border node with Purple robots. */
R5a: $\emptyset^{\phi}[?](P)(\emptyset^{\phi-1}) :: \rightarrow$

R5b-1: $\emptyset^{\phi}[B!](R!)(\emptyset^{\phi-1}) :: P$

R5b-2: $\emptyset^{\phi}[B!]\begin{pmatrix}R\\ B\end{pmatrix}(\emptyset^{\phi-1}) :: P$

R5b-3: $\emptyset^{\phi}\begin{bmatrix}R\\ [B]\end{bmatrix}(R!)(\emptyset^{\phi-1}) :: P$

The formal description of the algorithm is in Algorithm 2. Figure 5 illustrates an execution example of Algorithm 2. This figure assumes $\phi = 3$. Figure 5(a) shows an initial configuration. First, the left border White robot changes its light to Red by R1 (Fig. 5(b)). Next, the left border Red robot moves by R2a-1 (Fig. 5(c)). Note that, here, the robot does not change its light. In the next movement, the left border Red robot moves to a node with a White robot by R2b, and hence it changes its light to Blue (Fig. 5(d)). Then the left border White robot changes its light to Blue (Fig. 5(e)). Left border Blue robots can move by R3a-1. In Fig. 5(f), one of them completes the movement. In this case, another robot can move by R3a-2 (Fig. 5(g)). Next, right border White robots change their lights by R1 (Fig. 5(h)). After that, left and right border robots move toward each other by R2a-1 and R3a-1. In Fig. 5(i), some Blue and Red robots meet at a node but border robots continue to move until the number of occupied nodes is at most two by R2a-2 and R3a-2 (Fig. 5(j)). After the number of occupied nodes is at most two, some robots change their lights to Purple. In this case, the left Blue robot changes its light by R5b-2 (Fig. 5(k)). After that, all robots move to the node with a Purple robot by R5a and achieve gathering (Fig. 5(l)).

We now discuss the correctness of Algorithm 2. As with Algorithm 1, since there is no rule that non-border robot can execute by the definition of Algorithm 2, the following lemma holds.

▶ **Lemma 10.** *When a robot r_i looks, if it is a non-border robot, it cannot execute any action.*

To discuss the correctness, we change the definition of mega-cycle as follows: We consider the time instants such that the number of occupied nodes with White robots among non-border nodes (denoted as $\#O_W$) has just reduced at least one. That is, mega-cycles end

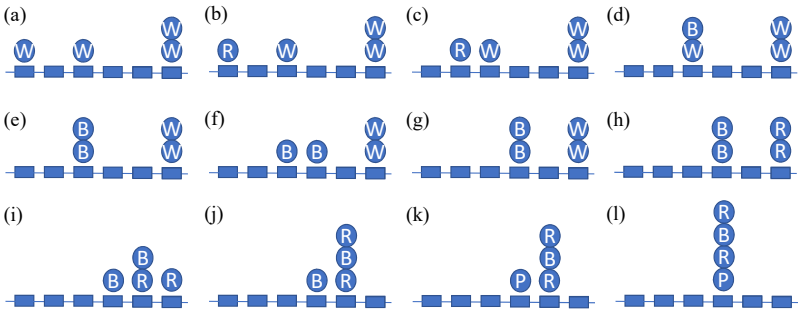


Figure 5 An execution example of Algorithm 2 with $\phi = 3$.

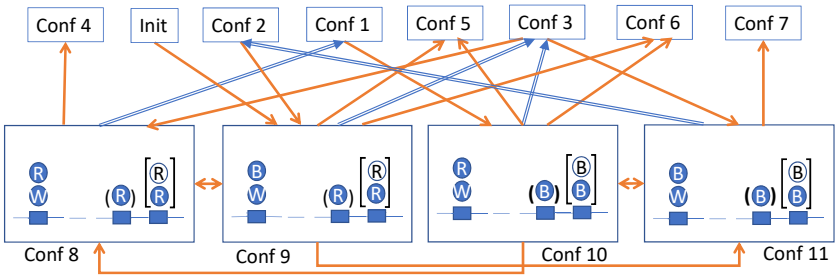


Figure 6 Transition Diagram for Algorithm 2 while $D > 2$.

when either the position of the border node moves to the nearest occupied node with White robots. If the position of the border node moves to the nearest occupied node with White robots, we say *the border absorbs White robots*.

Figure 6 shows a transition diagram of configurations for every mega-cycle. The doubly (resp. singly) lined arrows represent that $\#O_W$ is decreased by 2 (resp. 1). In the diagram, Init and Conf 1-7 are the same as those of Algorithm 1 and they have the same transitions between them as shown in Figure 3, where note that each node with W circle contains at least one White robot. In addition to these configurations, there exist four configurations Conf 8-11 in Algorithm 2. In Conf 1-3, both borders absorb White robots. In Conf 4-7, when one border absorbs White robots, the other border node is neighbored to an occupied node with a White robot. On the other hand, in Conf 8-11, when one border absorbs White robots, the other border node is not neighbored to any occupied node with a White robot.

The following lemma can be proved similarly to the proof of Lemma 3.

▶ **Lemma 11.** *Assume that a border node u is singly-colored with Red (resp. Blue) at time t and there is no robot in u at time $t'(> t)$. If the neighboring node of u (denoted as v) is an occupied node with a White robot at t, all robots in u change their color to Blue (resp. Red), move to v, and $\#O_W$ and D are reduced by at least one at t'. Otherwise, that is, when v is a node without White robots at t, all robots move to v and do not change their color and D is reduced by at least one at t'.*

By Lemma 11, each border node moves to the occupied node until either border node absorbs White robots in any mega-cycle. Transitions among Init and Conf 1-7 can be proved similarly to those in the corresponding lemmas using Lemma 11 instead of Lemma 3. The difference occurs when one border absorbs White robots and the neighboring node of the other border node is without White robots. In this case, since robots on the other border nodes moves to the neighboring node, the configuration becomes Conf 8-11. The proofs can be done similarly.

Also, we showed that the transition diagram in Figure 6 is correct while $D > 2$. Then, in each mega-cycle, because the number of occupied nodes between two borders decreases by at least one, the following lemma holds.

▶ **Lemma 12.** *From the initial configuration, $\#O_W$ decreases monotonically and eventually becomes at most one.*

▶ **Lemma 13.** *Let h be the number of occupied nodes where an original border robot r_h absorbed White robots in $\mathcal{G}'$ from the initial configuration and let u_h denote the current border node r_h is located. If h is odd (resp. even), r_h's light is Blue (resp. Red) when r_h comes into u_h.*

It can be easily verified by Figures 3 and 6 and Lemmas 12–13 that the following configurations occur when $\#O_W$ becomes at most one.

1. Conf 1, Conf 2, Conf 5 and Conf 6 ($D \geq 2$ and $\#O_W = 1$).
2. Conf 3 ($D \geq 1$ and $\#O_W = 0$) and Conf 9 ($D \geq 2$ and $\#O_W = 0$), and Conf 10 ($D \geq 2$ and $\#O_W = 0$).

In the former case, for configurations Conf 1,2,5,6($D = 2$ and $\#O_W = 1$), we have the following lemma, where Conf RW-MB and Conf BW-MR have been defined in the proof for Algorithm 1. In Conf 3($D = 1$), both border nodes may contain White robots with outdated views and contain no White robots.

▶ **Lemma 14.**
1. *Conf 1($D = 2$ and $\#O_W = 1$) $\rightarrow^*$ Conf RW-MB, or Conf 3($D = 1$)*
2. *Conf 2($D = 2$ and $\#O_W = 1$) $\rightarrow^*$ Conf BW-MR, or Conf 3($D = 1$)*
3. *Conf 5($D = 2$ and $\#O_W = 1$) $\rightarrow^*$ Conf BW-MR, or Conf 3($D = 1$)*
4. *Conf 6($D = 2$ and $\#O_W = 1$) $\rightarrow^*$ Conf RW-MB, or Conf 3($D = 1$)*

Otherwise ($D \geq 3$), we have the following transitions.

▶ **Lemma 15.**
1. *Conf 1($D \geq 3$ and $\#O_W = 1$) $\xrightarrow{MC}{}^*$ Conf 3($D \geq 1$ and $\#O_W = 0$), or Conf 10($D \geq 2$ and $\#O_W = 0$)*
2. *Conf 2($D \geq 3$ and $\#O_W = 1$)$\xrightarrow{MC}{}^*$ Conf 3($D \geq 1$ and $\#O_W = 0$), or Conf 9($D \geq 2$ and $\#O_W = 0$)*
3. *Conf 5($D \geq 3$ and $\#O_W = 1$)$\xrightarrow{MC}{}^*$ Conf 3($D \geq 1$ and $\#O_W = 0$), or Conf BW-MR($D = 1$ and $\#O_W = 0$)*
4. *Conf 6($D \geq 3$ and $\#O_W = 1$)$\xrightarrow{MC}{}^*$ Conf 3($D \geq 1$ and $\#O_W = 0$), or Conf RW-MB($D = 1$ and $\#O_W = 0$)*

We can prove that Conf BW-MR, Conf RW-MB and Conf 3($D = 1$) become gathering configuration in the following lemma.

▶ **Lemma 16.** *From configurations Conf BW-MR, Conf RW-MB and Conf 3($D = 1$), gathering is achieved.*

Then by Lemmas 14-16, it is sufficient to consider Conf 3($D \geq 2$ and $\#O_W = 0$), Conf 9($D \geq 2$ and $\#O_W = 0$), and Conf 10($D \geq 2$ and $\#O_W = 0$) for the former case.

The latter case has the following transitions. Note that, these transition does not reduce $\#O_W$ and just reduces the distance between the two border nodes. Note also that, the destinations of these transitions do not contain any White robots.

▶ **Lemma 17.**
1. *Conf 3 ($D \geq 3$ and $\#O_W = 0$) $\rightarrow^*$ Conf 3(($D = 2$ and $\#O_W = 0$) or Conf RW-BW*
2. *Conf 3($D \geq 3$ and $\#O_W = 0$)$\rightarrow^*$ Conf 10($D = 2$ and $\#O_W = 0$)*
3. *Conf 3($D \geq 3$ and $\#O_W = 0$)$\rightarrow^*$ Conf 9($D = 2$ and $\#O_W = 0$)*

Since Conf 9($D \geq 2$ and $\#O_W = 0$)(resp. Conf 10($D \geq 2$ and $\#O_W = 0$)) becomes Conf 3(($D = 2$ and $\#O_W = 0$) or Conf RW-BW, or Conf 9($D = 2$ and $\#O_W = 0$) (resp. Conf 9($D = 2$ and $\#O_W = 0$)), the correctness proof completes if we can show that these three configurations become gathering ones.

▶ **Lemma 18.** *Conf 3($D = 2$ and $\#O_W = 0$), Conf 9($D = 2$ and $\#O_W = 0$), and Conf 10($D = 2$ and $\#O_W = 0$) become gathering configurations.*

By the above discussion, we can derive the following theorem.

▶ **Theorem 19.** *Gathering is solvable in full-light of 4 colors when M_{init} is even and O_{init} is odd.*

It is an interesting open question whether gathering is solvable or not in full-light of 3 colors when M_{init} is even and O_{init} is odd.

4 Discussion

In this section, we discuss the gathering problem in other cases. First, we consider the case that M_{init} and O_{init} are even.

▶ **Definition 20.** *A configuration is* edge-view-symmetric *if there exist at least two distinct nodes hosting each at least one robot, and an edge (u_i, u_{i+1}) such that, for any integer $k \geq 0$, and for any robot r_1 at node u_{i-k}, there exists a robot r_2 at node u_{i+k+1} such that $\mathcal{V}_1 = \mathcal{V}_2$.*

▶ **Theorem 21.** *Deterministic gathering is impossible from any edge-view-symmetric configuration.*

▶ **Corollary 22.** *Starting from a configuration where M_{init} is even and O_{init} is even, and all robots have the same color, there exist initial configurations (e.g. edge-view-symmetric configurations) that a deterministic algorithm cannot gather.*

We now study the impact of an important property our algorithms satisfy: cautiousness [1].

▶ **Definition 23.** *A gathering algorithm is* cautious *if, in any execution, the direction to move is only toward other occupied nodes, i.e., robots are not adventurous and do not want to expand the covered area.*

Note that, the algorithms we provide in previous sections, only border robots move, and they only move toward occupied other nodes, hence our algorithms are cautious.

▶ **Lemma 24.** *A cautious algorithm that starts from a configuration with more than two border nodes cannot solve gathering.*

▶ **Lemma 25.** *A cautious algorithm that starts from a configuration with no border node cannot solve gathering.*

▶ **Theorem 26.** *For M_{init} even and O_{init} even, there exists no cautious gathering algorithm with $\phi = 1$, even when the initial configuration is* not *edge-view-symmetric.*

5 Conclusion

We presented the first gathering algorithms for myopic luminous robots in rings. One algorithm considers the case where M_{init} is odd, while the other is for the case where O_{init} is odd. The hypotheses used for our algorithms closely follow the impossibility results found for the other cases. Some interesting questions remain open:

- Are there any deterministic algorithms for the case where M_{init} and O_{init} are even (such solutions would have to avoid starting or ending up in an edge-view-symmetric situation)?
- Are there any algorithms for the case where M_{init} (resp. O_{init}) is odd that use fewer colors than ours? The current lower bound for odd M_{init} (resp. O_{init}) is 2 (resp. 3), but our solutions use 3 (resp. 4) colors.

References

1. Z. Bouzid, M. G. Potop-Butucaru, and S. Tixeuil. Optimal Byzantine-resilient convergence in uni-dimensional robot networks. *Theor. Comput. Sci.*, 411(34-36):3154–3168, 2010.
2. Q. Bramas, S. Devismes, and P. Lafourcade. Infinite Grid Exploration by Disoriented Robots. In *SIROCCO*, 2019.
3. G. D'Angelo, A. Navarra, and N. Nisse. A unified approach for gathering and exclusive searching on rings under weak assumptions. *Distributed Computing*, 30(1):17–48, 2017.
4. G. D'Angelo, G. D. Stefano, R. Klasing, and A. Navarra. Gathering of robots on anonymous grids and trees without multiplicity detection. *Theor. Comput. Sci.*, 610:158–168, 2016.
5. G. D'Angelo, G. D. Stefano, and A. Navarra. Gathering on rings under the Look-Compute-Move model. *Distributed Computing*, 27(4):255–285, 2014.
6. S. Das, P. Flocchini, G. Prencipe, N. Santoro, and M. Yamashita. Autonomous mobile robots with lights. *Theor. Comput. Sci.*, 609:171–184, 2016.
7. A. K. Datta, A. Lamani, L. L. Larmore, and F. Petit. Enabling Ring Exploration with Myopic Oblivious Robots. In *IPDPS*, pages 490–499, 2015.
8. P. Flocchini, G. Prencipe, and N. Santoro, editors. *Distributed Computing by Mobile Entities, Current Research in Moving and Computing*, volume 11340 of *LNCS*. Springer, 2019.
9. S. Guilbault and A. Pelc. Gathering asynchronous oblivious agents with local vision in regular bipartite graphs. *Theor. Comput. Sci.*, 509:86–96, 2013.
10. S. Guilbault and A. Pelc. Gathering Asynchronous Oblivious Agents with Restricted Vision in an Infinite Line. In *SSS*, volume 8255, pages 296–310, 2013.
11. T. Izumi, T. Izumi, S. Kamei, and F. Ooshita. Time-Optimal Gathering Algorithm of Mobile Robots with Local Weak Multiplicity Detection in Rings. *IEICE Transactions*, 96-A(6):1072–1080, 2013.
12. S. Kamei, A. Lamani, and F. Ooshita. Asynchronous Ring Gathering by Oblivious Robots with Limited Vision. In *WSSR*, pages 46–49, 2014.
13. S. Kamei, A. Lamani, F. Ooshita, S. Tixeuil, and K. Wada. Gathering on Rings for Myopic Asynchronous Robots with Lights. *arXiv:1911.04757*, 2019.
14. R. Klasing, A. Kosowski, and A. Navarra. Taking advantage of symmetries: Gathering of many asynchronous oblivious robots on a ring. *Theor. Comput. Sci.*, 411(34-36):3235–3246, 2010.
15. G. A. D. Luna and G. Viglietta. Robots with Lights. In *Distributed Computing by Mobile Entities, Current Research in Moving and Computing.*, volume 11340 of *LNCS*, pages 252–277. Springer, 2019.
16. S. Nagahama, F. Ooshita, and M. Inoue. Ring Exploration of Myopic Luminous Robots with Visibility More than One. In *SSS*, 2019.
17. F. Ooshita and S. Tixeuil. Ring Exploration with Myopic Luminous Robots. In *SSS*, pages 301–316, 2018.

18 I. Suzuki and M. Yamashita. Distributed Anonymous Mobile Robots: Formation of Geometric Patterns. *SIAM Journal on Computing*, 28(4):1347–1363, 1999.

19 G. Viglietta. Rendezvous of Two Robots with Visible Bits. In *ALGOSENSOR*, pages 291–306, 2013.

Optimal Register Construction in M&M Systems

Vassos Hadzilacos
Department of Computer Science, University of Toronto, Canada
vassos@cs.toronto.edu

Xing Hu
Department of Computer Science, University of Toronto, Canada
xing@cs.toronto.edu

Sam Toueg
Department of Computer Science, University of Toronto, Canada
sam@cs.toronto.edu

Abstract

Motivated by recent distributed systems technology, Aguilera *et al.* introduced a hybrid model of distributed computing, called *message-and-memory model* or *m&m model* for short [1]. In this model, processes can communicate by message passing and also by accessing some shared memory. We consider the basic problem of implementing an atomic single-writer multi-reader (SWMR) register shared by *all* the processes in m&m systems. Specifically, we give an algorithm that implements such a register in m&m systems and show that it is optimal in the number of process crashes that it can tolerate. This generalizes the well-known implementation of an atomic SWMR register in a pure message-passing system [4].

2012 ACM Subject Classification Theory of computation → Concurrency; Theory of computation → Parallel computing models; Theory of computation → Distributed computing models

Keywords and phrases asynchronous distributed system, shared memory, message passing

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.28

Funding This research was partially funded by the Natural Sciences and Engineering Research Council of Canada.

1 Introduction

Motivated by recent distributed systems technology [9, 12, 13, 19, 22], Aguilera *et al.* introduced a hybrid model of distributed computing, called *message-and-memory model* or *m&m model* for short [1]. In this model processes can communicate by message passing and also by accessing some shared memory. Since it is impractical to share memory among *all* processes in large distributed systems [8, 14, 15, 24], the m&m model allows us to specify which subsets of processes share which sets of registers. Among other results, Aguilera *et al.* show that it is possible to leverage the advantages of the two communication mechanisms (message-passing and share-memory) to improve the fault-tolerance of randomized consensus algorithms compared to a pure message-passing system [1].

In this paper, we consider the more basic problem of implementing an atomic single-writer multi-reader (SWMR) register shared by *all* the processes in m&m systems, and we give an algorithm that is optimal in the number of process crashes that it can tolerate. This generalizes the well-known implementation of an atomic SWMR register in a pure message-passing system [4]. We now describe our results in more detail.

A *general* m&m system $\mathcal{S}_L$ is specified by a set of n asynchronous processes that can send messages to each other over asynchronous reliable links, and by a collection $L = \{S_1, S_2, \ldots, S_m\}$ where each S_i is a subset of processes: for each S_i, there is a set of atomic registers that can be shared by processes in S_i and only by them. Even though

© Vassos Hadzilacos, Xing Hu, and Sam Toueg;
licensed under Creative Commons License CC-BY
23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller; Article No. 28; pp. 28:1–28:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

the m&m model allows the collection L to be arbitrary, in practice hardware technology imposes a structure on L [8, 14]: for processes to share memory, they must establish a connection between them (e.g., an RDMA connection). These connections are naturally modelled by an undirected *shared-memory graph* G whose nodes are the processes and whose edges are shared-memory connections [1]. Such a graph G defines what Aguilera *et al.* call a *uniform* m&m system $\mathcal{S}_G$, where each process has atomic registers that it can share with its neighbours in G (and only with them). Note that $\mathcal{S}_G$ is the instance of the general m&m system $\mathcal{S}_L$ with $L = \{S_1, S_2, \ldots, S_n\}$ where each S_i consists of a process and its neighbours in G. Furthermore, if G is the trivial graph with n nodes but no edges, the m&m system $\mathcal{S}_G$ that G induces is just a pure message-passing asynchronous system with n processes.

We consider the implementation of an atomic SWMR register $\mathbf{R}$, shared by all the processes, in both general and uniform m&m systems. For each general m&m system $\mathcal{S}_L$, we determine the maximum number of crashes t_L for which it is possible to implement $\mathbf{R}$ in $\mathcal{S}_L$: we give an algorithm that tolerates t_L crashes and prove that no algorithm can tolerate more than t_L crashes. Similarly, for each shared-memory graph G and its corresponding uniform m&m system $\mathcal{S}_G$, we use the topology of G to determine the maximum number of crashes t_G for which it is possible to implement $\mathbf{R}$ in $\mathcal{S}_G$. By specifying t_G in terms of the topology of G, one can leverage results from graph theory to design m&m systems that can implement $\mathbf{R}$ with high fault tolerance and relatively few RDMA connections per process. For example, it allows us to design an m&m system with 50 processes that can implement a *wait-free* $\mathbf{R}$ (i.e., this implementation can tolerate *any* number of process crashes) with only 7 RDMA connections per process; as explained in Section 4, this is optimal in some precise sense.

An important remark is now in order. In this paper we consider RDMA systems where process crashes do not affect the accessibility of the shared registers of that system. This is the case in systems where the CPU, the DRAM (main memory), and the NIC (Network Interface Controller) are separate entities: for example, in the InfiniBand cluster evaluated in [21], the crash of a CPU, and of the processes that it hosts, does not prevent other processes from accessing its DRAM because it can use the NIC without involving the CPU; see also [8, 10, 26].

2 Model outline

We consider m&m systems with a set of n asynchronous processes $\Pi = \{p_1, p_2, \ldots p_n\}$ that may *crash*. To define these systems we first recall the definition of atomic SWMR registers.

2.1 Atomic SWMR registers

A SWMR register R shared by a set S of processes is a register that can be written (sequentially) by exactly one process $w \in S$ and can be read by all processes in S; we say that w is the *writer* of R [18].

We now define an *atomic* SWMR register [4, 18] in terms of two simple properties that they must satisfy. To do so, we first define what it means for a (read or write) operation to precede another operation, and for two operations to be concurrent. We say that an operation o *precedes* another operation o' if and only if o completes before o' starts. A write operation o *immediately precedes a read operation* r if and only if o precedes r, and there is no write operation o' such that o precedes o' and o' precedes r. Operations o and o' are *concurrent* if and only if neither precedes the other.

We assume, without loss of generality, that the values successively written by the single writer w of a SWMR register R are distinct, and different from the initial value of R.[1] Let v_0 denote the *initial value* of R, and v_k denote the value written by the k-th write operation of w. A SWMR register R is *atomic* if and only if it satisfies the following two properties:

▶ **Property 1.** *If a read operation r returns the value v then:*

- *there is a write v operation that immediately precedes r or is concurrent with r, or*
- *no write operation precedes r and $v = v_0$.*

▶ **Property 2.** *If two read operations r and r' return values v_k and $v_{k'}$, respectively, and r precedes r', then $k \leq k'$.*

2.2 General m&m systems

Let $L = \{S_1, S_2, \ldots, S_m\}$ be any bag of non-empty subsets of $\Pi = \{p_1, p_2, \ldots p_n\}$.

▶ **Definition 3.** *$\mathcal{M}_L$ is the class of m&m systems (induced by L), each consisting of:*

1. *The processes in Π.*
2. *Reliable asynchronous communication links between every pair of processes in Π.*
3. *The following set of registers: For each subset of processes S_i in L, a non-empty set of atomic registers that are shared by the processes in S_i (and only by them).*

Note that $\mathcal{M}_L$ includes m&m systems that differ by the type and number of registers shared by the processes in each S_i; for example they could be sharing multi-writer multi-reader atomic registers.

Since we are interested in implementing atomic SWMR registers (shared by *all* processes in the system), here we focus on an m&m system of $\mathcal{M}_L$ in which the set of registers shared by the processes in each set S_i are atomic SWMR registers. More precisely, we focus on the m&m system $\mathcal{S}_L$ defined below:

▶ **Definition 4.** *The general m&m system $\mathcal{S}_L$ (induced by L) consists of:*

1. *The processes in Π.*
2. *Reliable asynchronous communication links between every pair of processes in Π.*
3. *The following set of registers: For each subset of processes S_i in L and each process $p \in S_i$, an atomic SWMR register, denoted $R_i[p]$, that can be written by p and read by all processes in S_i (and only by them).*

In this paper, for every L, we give an algorithm that implements atomic SWMR registers shared by all processes in the m&m system $\mathcal{S}_L$, and we show that this algorithm is optimal in the number of process crashes that can be tolerated. In fact we prove that any algorithm that implements such registers in *any* m&m system in $\mathcal{M}_L$, (not only in $\mathcal{S}_L$) cannot tolerate more crashes. This justifies our focus on $\mathcal{S}_L$: considering members of $\mathcal{M}_L$ with more or stronger registers than $\mathcal{S}_L$ does not improve the fault tolerance of implementing atomic SWMR registers shared by all.

Without loss of generality we assume the following:

▶ **Assumption 5.** *The bag $L = \{S_1, S_2, \ldots, S_m\}$ of subsets of $\Pi = \{p_1, p_2, \ldots p_n\}$ is such that every process in Π is in at least one of the subsets S_j of L.*

[1] This can be ensured by the writer w writing values of the form $\langle sn, v\rangle$ where sn is the value of a counter that w increments before each write.

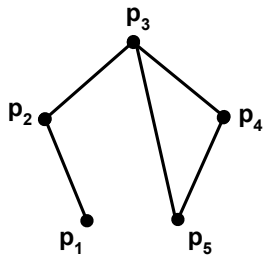


Figure 1 A graph G.

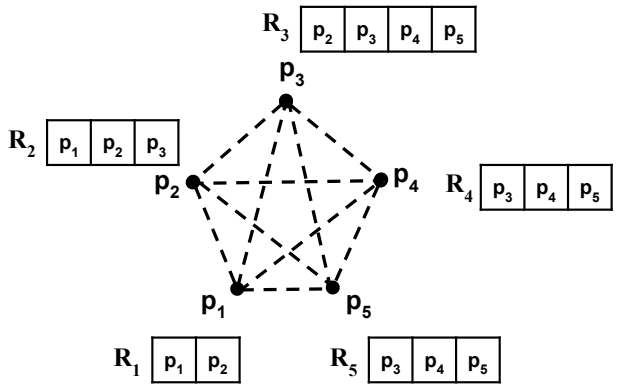


Figure 2 The uniform m&m system $\mathcal{S}_G$ induced by G.

This assumption can be made without loss of generality because it does not strengthen the system $\mathcal{S}_L$ induced by L. In fact, given a bag L that does *not* satisfy the above assumption, we can construct a bag that satisfies the assumption as follows: for every process p_i in Π that is *not* contained in any S_j of L, we can add the singleton set $\{p_i\}$ to L. Let L' be the resulting bag. By Definition 4(3) above, adding $\{p_i\}$ to L results in adding only a *local* register to the induced system $\mathcal{S}_L$, namely, an atomic register that p_i (trivially) shares only with itself. So $\mathcal{S}_{L'}$ is just $\mathcal{S}_L$ with some additional local registers. Note that a pure message-passing system (with no shared memory) with n processes $p_1, p_2, \ldots p_n$ is modelled by the system $\mathcal{S}_L$ where $L = \{\{p_1\}, \{p_2\}, \ldots \{p_n\}\}$.

2.3 Uniform m&m systems

Let $G = (V, E)$ be an undirected graph such that $V = \Pi$, i.e., the nodes of G are the n processes $p_1, p_2, \ldots, p_n$ of the system. For each $p_i \in V$, let $N(p_i) = \{p_j \mid (p_i, p_j) \in E\}$ be the *neighbours* of p_i in G, and let $N^+(p_i) = N(p_i) \cup \{p_i\}$.

▶ **Definition 6.** *The uniform m&m system $\mathcal{S}_G$ (induced by G) is the m&m system $\mathcal{S}_L$ where $L = \{S_1, S_2, \ldots, S_n\}$ with $S_i = N^+(p_i)$.*[2]

The graph G induces the uniform m&m system $\mathcal{S}_G$ where processes can communicate by message passing (via reliable asynchronous communication links), and also by shared memory as follows: for each process p_i, and every neighbour p of p_i in G (including p_i) there is an atomic SWMR register $R_i[p]$ that can be written by p and read by every neighbour of p_i in G (including p_i). We can think of the registers $R_i[*]$ as being physically located in the DRAM of the host of p_i, and every neighbour of p_i accessing these registers over its RDMA connection to p_i (which is modelled by an edge of G).[3]

For example, in Figures 1 and 2 we show a graph G and the uniform m&m system $\mathcal{S}_G$ induced by G. Here G has five nodes representing processes p_1, p_2, p_3, p_4, p_5; the edges of G represent the RDMA connections that allow these processes to share registers. The uniform m&m system $\mathcal{S}_G$ induced by G is the system $\mathcal{S}_L$ for $L = \{S_1, S_2, S_3, S_4, S_5\}$ where

[2] Note that L satisfies Assumption 5 because each $S_i = N^+(p_i)$ contains p_i.

[3] As we mentioned in the introduction, we assume that the crash of p_i does not prevent the neighbours of p_i from accessing the shared registers $R_i[*]$.

each S_i consists of p_i and its neighbours in G: specifically, $S_1 = \{p_1, p_2\}, S_2 = \{p_1, p_2, p_3\}$, $S_3 = \{p_2, p_3, p_4, p_5\}$, and $S_4 = S_5 = \{p_3, p_4, p_5\}$. The box adjacent to each process p_i in $\mathcal{S}_G$ represents the atomic SWMR registers that are shared among p_i and its neighbours in G (intuitively these registers are located at p_i). For example, in the box adjacent to process p_2, the component labelled p_1 represents the register $R_2[p_1]$ that can be written by p_1 and read by all the neighbours of p_2 in G, namely p_1, p_2, and p_3. Similarly, registers $R_2[p_2]$ and $R_2[p_3]$ can be written by p_2 and p_3, respectively, and read by p_1, p_2, and p_3. The dashed lines in Figure 2 represent the asynchronous message-passing links between the processes of $\mathcal{S}_G$.

3 Atomic SWMR register implementation in general m&m systems

Let $\mathcal{S}_L$ be the general m&m system induced by a bag $L = \{S_1, \ldots, S_m\}$ of subsets of $\Pi = \{p_1, p_2, \ldots p_n\}$. Recall that in system $\mathcal{S}_L$, for every S_i in L, the processes in S_i share some atomic SWMR registers that can be read *only* by the processes in S_i (recall that it is impractical to share registers among *all* processes in large distributed systems [8, 14, 15, 24]). In the rest of the paper, we determine the maximum number of process crashes t_L that may occur in $\mathcal{S}_L$ such that it is possible to *implement* in $\mathcal{S}_L$ a shared atomic SWMR register readable by *all* processes in $\mathcal{S}_L$. Intuitively, if $t \leq t_L$ processes may crash, then any two subsets of processes of size $n - t$ either intersect, or they each contain a process that can communicate with the other via a shared SWMR register that it can write and the other can read. If $t > t_L$ processes may crash, then there are two subsets of processes of size $n - t$ that are disjoint and cannot communicate via shared SWMR register.

▶ **Definition 7.** *Given a bag $L = \{S_1, \ldots, S_m\}$ of subsets of $\Pi = \{p_1, p_2, \ldots p_n\}$, t_L is the maximum integer t such that the following condition holds: For all disjoint subsets P and P' of Π of size $n - t$ each, some set S_i in L contains both a process in P and a process in P'.*

Note that if $t \leq \lfloor (n-1)/2 \rfloor$ then there are *no* disjoint subsets P and P' of Π of size $n-t$ each, and so the above condition is vacuously true. Therefore $t_L \geq \lfloor (n-1)/2 \rfloor$. Recall that for a pure message-passing system, $L = \{\{p_1\}, \{p_2\}, \ldots \{p_n\}\}$, so in this system $t_L = \lfloor (n-1)/2 \rfloor$.

To illustrate Definition 7, suppose $\Pi = \{p_1, p_2, p_3, p_4, p_5\}$ and $L = \{S_1, S_2, S_3\}$ where $S_1 = \{p_1, p_2\}$, $S_2 = \{p_4, p_5\}$, and $S_3 = \{p_2, p_4, p_3\}$. By the definition of t_L: (1) $t_L \geq 3$ because for any two disjoint subsets of Π of size $5 - 3 = 2$ each, there exists a set S_i in L that intersects both subsets; e.g., for subsets $\{p_1, p_5\}$ and $\{p_3, p_4\}$, the set $S_2 = \{p_4, p_5\}$ intersects both of them. (2) $t_L < 4$ because there are two disjoint subsets $\{p_1\}, \{p_5\}$ of size $5 - 4 = 1$ each, such that no set S_i in L contains both p_1 and p_5. So in this example $n = 5$ and $t_L = 3 > \lfloor (5-1)/2 \rfloor = 2$.

We now prove that in the general m&m system $\mathcal{S}_L$, it is possible to implement an atomic SWMR register readable by *all* processes *if and only if* at most t_L processes may crash in $\mathcal{S}_L$. More precisely:

▶ **Theorem 8.** *Let $\mathcal{S}_L$ be the general m&m system induced by a bag $L = \{S_1, \ldots, S_m\}$ of subsets of $\Pi = \{p_1, p_2, \ldots p_n\}$.*

- *If at most t_L processes crash in $\mathcal{S}_L$, then for every process w in $\mathcal{S}_L$, it is possible to implement an atomic SWMR register writable by w and readable by **all** processes in $\mathcal{S}_L$.*
- *If more than t_L processes crash in $\mathcal{S}_L$, then for some process w in $\mathcal{S}_L$, it is impossible to implement an atomic SWMR register writable by w and readable by **all** processes in $\mathcal{S}_L$.*

The above theorem is a direct corollary of Theorem 18 (Section 3.1) and Theorem 19 (Section 3.2).

3.1 Algorithm

We now show how to implement an atomic SWMR register $\mathbf{R}$, that can be written by an arbitrary fixed process w and read by *all* processes, in an m&m system $\mathcal{S}_L$ where at most t_L processes may crash. This implementation is given in terms of the procedures WRITE() and READ() shown in Algorithm 1.

Without loss of generality we assume that for all $i \geq 1$, the i-th value that the writer writes is of the form $\langle i, val \rangle$, and the initial value of the register $\mathbf{R}$ is $\langle 0, u_0 \rangle$. To write $\langle i, val \rangle$ into $\mathbf{R}$, the writer w calls the procedure WRITE($\langle i, val \rangle$). To read $\mathbf{R}$, a process q calls the procedure READ() which returns a value of the form $\langle i, val \rangle$. The sequence number i makes the values written to $\mathbf{R}$ unique.

Algorithm 1 Implementation of an atomic SWMR register writeable by process w and readable by all processes in $\mathcal{S}_L$, provided that at most t_L processes crash.

SHARED VARIABLES

For all S_i in L and all p in S_i:

$R_i[p]$: atomic SWMR register writeable by p and readable by every process in $S_i \in L$; initialized to $\langle 0, u_0 \rangle$.

WRITE($\langle sn_w, u \rangle$): ▷ executed by the writer w

send $\langle \text{W}, \langle sn_w, u \rangle \rangle$ **to every process** p in $\mathcal{S}_L$
wait for $\langle \text{ACK-W}, sn_w \rangle$ **messages from** $n - t_L$ **distinct processes**
return

▷ executed by every process p in $\mathcal{S}_L$

upon receipt of a $\langle \text{W}, \langle sn_w, u \rangle \rangle$ **message from process** $\boldsymbol{w}$:
  for every i in $\{1, \ldots, m\}$ such that $p \in S_i$ **do**
    $\langle sn, - \rangle \leftarrow R_i[p]$
    if $sn_w > sn$ **then**
      $R_i[p] \leftarrow \langle sn_w, u \rangle$
  send $\langle \text{ACK-W}, sn_w \rangle$ **to process** w

READ(): ▷ executed by any process q

$sn_r \leftarrow sn_r + 1$
send $\langle \text{R}, sn_r \rangle$ **to every process** p in $\mathcal{S}_L$
wait for $\langle \text{ACK-R}, sn_r, \langle -, - \rangle \rangle$ **messages from** $n - t_L$ **distinct processes**
$\langle seq, val \rangle \leftarrow \max\{\langle r_sn, r_u \rangle \mid \text{received a } \langle \text{ACK-R}, sn_r, \langle r_sn, r_u \rangle \rangle \text{ message}\}$
WRITE($\langle seq, val \rangle$)
return $\langle seq, val \rangle$

▷ executed by every process p in $\mathcal{S}_L$

upon receipt of a $\langle \text{R}, sn_r \rangle$ **message from a process** $\boldsymbol{q}$:
  $\langle r_sn, r_u \rangle \leftarrow \max\{\langle sn, u \rangle \mid \exists i \in \{1, \ldots, m\}, p \in S_i \text{ and } \exists p' \in S_i, R_i[p'] = \langle sn, u \rangle\}$
  send $\langle \text{ACK-R}, sn_r, \langle r_sn, r_u \rangle \rangle$ **to process** q

Algorithm 1 generalizes the well-known implementation of an atomic SWMR register in pure message-passing systems [4]. To write a new value into $\mathbf{R}$, the writer w sends messages to all processes asking them to write the new value into all the shared SWMR registers that they can write in $\mathcal{S}_L$. The writer then waits for acknowledgments from $n - t_L$ processes indicating that they have done so. To read $\mathbf{R}$, a process sends messages to all processes asking them for the most up-to-date value that they can find in all the shared SWMR registers that they can read. The reader waits for $n - t_L$ responses, selects the most up-to-date value among them, writes back that value (using the same procedure that the writer uses), and returns it. From the definition of t_L it follows that every write of $\mathbf{R}$ "intersects" with every read of $\mathbf{R}$ at some shared SWMR register of $\mathcal{S}_L$. Note that since at most t_L processes crash, the waiting mentioned above does not block any process.

We now show that the procedure WRITE(), called by the writer w, and the procedure READ(), called by any process q in $\mathcal{S}_L$, implement an atomic SWMR register $\mathbf{R}$. To do so, we show that the calls of WRITE() *by* w and of READ() by any process satisfy Properties 1 and 2 of atomic SWMR registers given in Section 2.1.

▶ **Definition 9.** *The operation write(v) is the execution of* WRITE(v) ***by the writer*** *w for some tuple $v = \langle sn_w, u\rangle$: this operation starts when w calls* WRITE(v) *and it completes if and when this call returns. An operation read(v) is an execution of* READ() *that returns v to some process q: this operation starts when q calls* READ() *and it completes when this call returns v to q.*

Let $v_0 = \langle 0, u_0\rangle$ be the initial value of the implemented register $\mathbf{R}$, and, for $k \geq 1$, let $v_k = \langle k, -\rangle$ denote the k-th value written by the writer w on $\mathbf{R}$. Note that all v_k's are distinct: for all $i \neq j \geq 0, v_i \neq v_j$.

Let $\mathcal{S}_L$ be the general m&m system induced by a bag $L = \{S_1, \ldots, S_m\}$ of subsets of $\Pi = \{p_1, p_2, \ldots p_n\}$. To prove the correctness of the SWMR implementation shown in Algorithm 1, we now consider an arbitrary execution of this implementation in $\mathcal{S}_L$ under the assumption that *at most t_L processes crash.*

▶ **Lemma 10.** *Any read($-$) or write($-$) operation executed by a process that does not crash completes.*

Proof. The only statements that could prevent the completion of a *read*($-$) or *write*($-$) operation are the **wait** statements of line 2 and line 12. But since communication links are reliable, these wait statements are for $n - t_L$ acknowledgements, and at most t_L processes out of the n processes of $\mathcal{S}_L$ may crash, it is clear that these wait statements cannot block. ◀

We first note that every read operation returns some v_k for $k \geq 0$.

▶ **Lemma 11.** *If r is a read(v) operation in the execution, then $v = v_k$ for some $k \geq 0$.*

Proof. This proof is straightforward and omitted here. ◀

The next lemma says that no read operation can read a "future" value, i.e., a value that is written *after* the read completes.

▶ **Lemma 12.** *If r is a read(v) operation in the execution, then either $v = v_0$, or $v = v_k$ such that the operation write(v_k) precedes r or is concurrent with r.*

Proof. This proof is straightforward and omitted here. ◀

Note that the guard in lines 6-8 (which is the only place where the shared SWMR registers are updated) ensures that the content of each shared SWMR register in $\mathcal{S}_L$ is non-decreasing in the following sense:

▶ **Observation 13.** *[Register monotonicity] For all $1 \leq i \leq m$ and all $p \in S_i$, if $R_i[p] = \langle k, -\rangle$ at some time t and $R_i[p] = \langle k', -\rangle$ at some time $t' \geq t$ then $k' \geq k$.*

▶ **Lemma 14.** *For all $k \geq 1$, if a call to the procedure* WRITE(v_k) *returns before a read(v) operation starts, then $v = v_\ell$ for some $\ell \geq k$.*

Proof. Suppose a call to WRITE(v_k) returns before a *read*(v) operation starts; we must show that $v = v_\ell$ with $\ell \geq k$. Note that before this call of WRITE(v_k) returns, $\langle$ACK-W$, k\rangle$ messages are received from a set P of $n - t_L$ distinct processes (see line 2 of the WRITE() procedure). From lines 5-8, which are executed before these $\langle$ACK-W$, k\rangle$ messages are sent, and by Observation 13, it is now clear that the following holds:

▶ **Claim 14.1.** *By the time* WRITE(v_k) *returns, every shared SWMR register in $\mathcal{S}_L$ that can be written by a process in P contains a tuple $\langle k', -\rangle$ with $k' \geq k$.*

Now consider the *read*(v) operation, say it is by process q. Recall that *read*(v) is an execution of the READ() procedure that returns v to q. When q calls READ(), it increments a local counter sn_r and asks every process p in $\mathcal{S}_L$ to do the following: (a) read every SWMR register that p can read, and (b) reply to q with a $\langle$ACK-R$, sn_r, \langle r_sn, r_u\rangle\rangle$ message such that $\langle r_sn, r_u\rangle$ is the tuple with the maximum r_sn that p read. By line 12 of the READ() procedure, q waits to receive such $\langle$ACK-R$, sn_r, \langle -, -\rangle\rangle$ messages from a set P' of $n - t_L$ distinct processes, and q uses these messages to select the value v as follows:

$$v \leftarrow \max\{(r_sn, r_u) \mid q \text{ received some } \langle \text{ACK-R}, sn_r, \langle r_sn, r_u\rangle\rangle \text{ from a process in } P'\}$$

Thus, by Lemma 11, it is clear that:

▶ **Claim 14.2.** *$v = v_\ell$ where $\ell = \max\{j \mid q$ received a $\langle$ACK-R$, sn_r, \langle j, -\rangle\rangle$ message from a process in $P'\}$.*

▶ **Claim 14.3.** *Some set S_i in L contains both a process in P and a process in P'.*

Proof of Claim 14.3. If P and P' are disjoint, the claim follows directly from Definition 7 of t_L. If P and P' intersect, let p be a process in both P and P'. By Assumption 5, p is in some set S_i in L, and the claim follows. ◀

By the above claim, some set S_i in L contains a process p in P and a process p' in P'. Since $p \in S_i$ and $p' \in S_i$, $R_i[p]$ is one of the SWMR registers that can be written by p and read by p'. From Claim 14.1, by the time the call to WRITE(v_k) returns, $R_i[p]$ contains a tuple $\langle k', -\rangle$ such that $k' \geq k$ (*). Since $p' \in P'$, during the execution of *read*(v) by q, p' reads all the shared SWMR registers that it can read, including $R_i[p]$. Since *read*(v) starts after WRITE(v_k) returns, p' reads $R_i[p]$ after WRITE(v_k) returns. Thus, by (*) and the monotonicity of $R_i[p]$ (Observation 13), p' reads from $R_i[p]$ a tuple $\langle r_sn, -\rangle$ with $r_sn \geq k' \geq k$. Then p' selects the tuple $\langle j, -\rangle$ with the maximum sn among all the $\langle sn, -\rangle$ tuples that it read (see line 17); note that $j \geq k$. So the $\langle$ACK-R$, sn_r, \langle j, -\rangle\rangle$ message that p' sends to q, and q uses to select v, is such that $j \geq k$. So, by Claim 14.2, $v = v_\ell$ such that $\ell \geq j \geq k$. ◀

Lemma 14 immediately implies the following:

▶ **Corollary 15.** *For all $k \geq 1$, if a write(v_k) operation precedes a read(v) operation then $v = v_\ell$ with $\ell \geq k$.*

We now show that Algorithm 1 satisfies Property 1 and 2 of atomic SWMR registers.

▶ **Lemma 16.** *The write$(-)$ and read$(-)$ operations satisfy Property 1.*

Proof. Suppose for contradiction that Property 1 does not hold. Thus there is a read operation $r = read(v)$ such that:

(a) there is no $write(v)$ operation that immediately precedes r or is concurrent with r, and
(b) some $write(-)$ operation precedes r, or $v \neq v_0$.

There are two cases.

1. $v = v_0$. By (b) above, some $write(-)$ operation, say $write(v_k)$, precedes r. Thus $write(v_k)$ precedes $read(v_0)$. Since $k \geq 1$ this contradicts Corollary 15.
2. $v \neq v_0$. By Lemma 12, $v = v_k$ such that the operation $write(v_k)$ precedes r, or $write(v_k)$ is concurrent with r. By (a) above, $write(v_k)$ does not *immediately* precede r, and $write(v_k)$ is not concurrent with r. Thus, $write(v_k)$ precedes, but not *immediately*, r. Let $write(v_{k'})$ be the write operation that immediately precedes r. Note that $write(v_k)$ precedes $write(v_{k'})$, so $k < k'$. Since $write(v_{k'})$ precedes $r = read(v)$, by Corollary 15, $v = v_\ell$ with $\ell \geq k'$, so $\ell > k$. This contradicts that $v = v_k$.

Since both cases lead to a contradiction, Property 1 holds. ◀

▶ **Lemma 17.** *The write$(-)$ and read$(-)$ operations satisfy Property 2.*

Proof. We have to show that if a $read(v_k)$ operation precedes a $read(v_{k'})$ operation then $k \leq k'$. Suppose $read(v_k)$ precedes $read(v_{k'})$. Note that during the $read(v_k)$ operation, namely in line 14, there is a call to the procedure $\textsc{Write}(v_k)$ which returns before the $read(v_k)$ operation completes. So this call to $\textsc{Write}(v_k)$ returns before the $read(v_{k'})$ operation starts. By Lemma 12, $k \leq k'$. ◀

Lemmas 10, 16 and 17 immediately imply:

▶ **Theorem 18.** *Let $\mathcal{S}_L$ be the general m&m system induced by a bag $L = \{S_1, \ldots, S_m\}$ of subsets of $\Pi = \{p_1, p_2, \ldots p_n\}$. If at most t_L processes crash in $\mathcal{S}_L$, for every process w in $\mathcal{S}_L$, Algorithm 1 implements an atomic SWMR register writable by w and readable by **all** processes in $\mathcal{S}_L$.*

3.2 Lower bound

▶ **Theorem 19.** *Let $\mathcal{S}_L$ be the general m&m system induced by a bag $L = \{S_1, \ldots, S_m\}$ of subsets of $\Pi = \{p_1, p_2, \ldots p_n\}$. If more than t_L processes crash in $\mathcal{S}_L$, for some process w in $\mathcal{S}_L$, there is no algorithm that implements an atomic SWMR register writable by w and readable by **all** processes in $\mathcal{S}_L$.*

Proof. Let $\mathcal{S}_L$ be the general m&m system induced by a bag $L = \{S_1, \ldots, S_m\}$ of subsets of $\Pi = \{p_1, p_2, \ldots p_n\}$. Suppose for contradiction that $t > t_L$ processes can crash in $\mathcal{S}_L$, but for every process w in $\mathcal{S}_L$, there is an algorithm $\mathcal{A}_w$ that implements an atomic SWMR register writable by w and readable by **all** processes in $\mathcal{S}_L$ (*).

Since $t > t_L$, by the Definition 7 of t_L there are two disjoint subsets P and P' of Π, of size $n - t$ each, such that: no set S_i in L contains both a process in P and a process in P' (**). Since P and P' are disjoint, the sets P, P', and $Q = \Pi - (P \cup P')$ form a partition of Π.

Let the writer w be any process in P. Let $\mathcal{A}$ be an algorithm that tolerates $t > t_L$ process crashes in $\mathcal{S}_L$ and implements an atomic SWMR register $\mathbf{R}$ that is writable by w and readable by **all** processes in $\mathcal{S}_L$; this algorithm exists by our initial assumption (*).

Since $|P \cup Q \cup P'| = n$, clearly $|P \cup Q| = |P' \cup Q| = n - (n - t) = t$. Since algorithm $\mathcal{A}$ tolerates t crashes, it works correctly in every execution in which all the processes in $P \cup Q$ or in $P' \cup Q$ crash.

We now define three executions E_1, E_2, and E_3 of algorithm $\mathcal{A}$. These are illustrated in Figure 3.

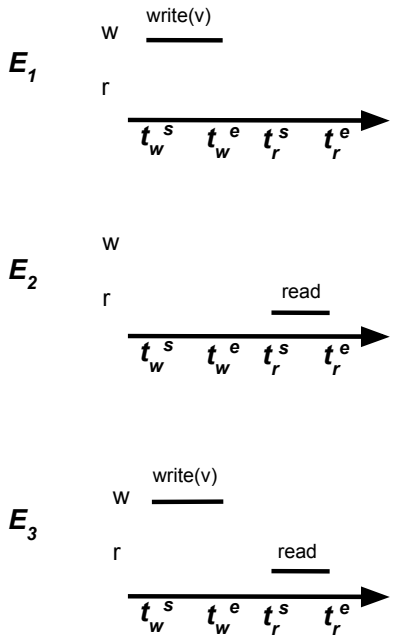


Figure 3 Scenarios for Theorem 19.

Execution E_1 of algorithm $\mathcal{A}$ is defined as follows:

- The processes in $P' \cup Q$ crash from the beginning of the execution; they take no steps in E_1.
- At some time t_w^s the writer w starts an operation to write the value v into the implemented register $\mathbf{R}$, for some $v \neq v_0$, where v_0 is the initial value of $\mathbf{R}$. Since the number of processes that crash in E_1 is $|P' \cup Q| = t$, and the algorithm $\mathcal{A}$ tolerates t crashes, this write operation eventually terminates, say at time t_w^e.
- After this write terminates, no process takes a step up to and including some time $t_r^s > t_w^e$.

Note that in E_1, processes in P are the only ones that take steps up to time t_r^s.

Execution E_2 of algorithm $\mathcal{A}$ is defined as follows:

- The processes in $P \cup Q$ crash from the beginning of the execution; they take no steps in E_2.
- At time t_r^s, some process $r \in P'$ starts a read operation on the implemented register $\mathbf{R}$. Since the number of processes that crash in E_2 is $|P \cup Q| = t$, and the algorithm $\mathcal{A}$ tolerates t crashes, this read operation terminates, say at time t_r^e.

Since no write operation precedes the read operation in E_2, Property 1 of atomic SWMR registers implies:

▶ **Claim 19.1.** *At time t_r^e in E_2 the read operation returns the initial value v_0 of $\boldsymbol{R}$.*

We now construct an execution E_3 of the algorithm $\mathcal{A}$ that merges E_1 and E_2, and contradicts the atomicity of the implemented $\mathbf{R}$. E_3 is identical to E_1 up to time t_r^s, and it is identical to E_2 from time t_r^s to t_r^e (note that in E_3 processes in Q can only take steps *after* time t_r^e). To obtain this merged run E_3, intuitively we delay the messages sent by processes in P to processes in P' to after time t_r^e, and we also use the fact that processes in P' cannot read any of the shared registers in $\mathcal{S}_L$ that processes in P may have written by time t_r^s (this is because of (**)).

▶ **Claim 19.2.** *There is an execution E_3 of algorithm $\mathcal{A}$ such that*

(a) *up to and including time t_w^e, E_3 is indistinguishable from E_1 to all processes.*

(b) *up to and including time t_r^e, E_3 is indistinguishable from E_2 to all processes in P'.*

(c) *No process crashes in E_3.*

Proof of Claim 19.2. Until time t_r^s, E_3 is identical to E_1. We now show that it is possible to extend E_3 in the time interval $[t_r^s, t_r^e]$ with the sequence of steps that the processes in P' executed during the same time interval in E_2.[4] More precisely, let $s^1, s^2, \ldots, s^\ell$ be the sequence of steps executed during the time interval $[t_r^s, t_r^e]$ in E_2. Since only processes in P' take steps in E_2, $s^1, s^2, \ldots, s^\ell$ are all steps of processes in P'. Let C_2^0 be the configuration of the system $\mathcal{S}_L$ at time t_r^s in E_2,[5] and let C_2^i be the configuration that results from applying step s^i to configuration C_2^{i-1}, for all i such that $1 \leq i \leq \ell$. We will prove that there are configurations $C_3^0, C_3^1, \ldots, C_3^\ell$ of $\mathcal{S}_L$ extending E_3 at time t_r^s such that:

(i) every process in P' has the same state in C_3^i as in C_2^i;

(ii) the set of messages sent by processes in P' to processes in P', but not yet received, is the same in C_3^i as in C_2^i;

(iii) every shared register readable by processes in P' has the same value in C_3^i as in C_2^i; and

(iv) if $i \neq 0$, C_3^i is the result of applying step s^i to configuration C_3^{i-1}.

This is shown by induction on i.

For the basis of the induction, $i = 0$, we take C_3^0 to be the configuration of the system just before time t_r^s in E_3. Since no process in P' takes a step before time t_r^s in either E_2 or E_3, C_3^0 satisfies properties (i) and (ii).

▶ **Claim 19.3.** *At time t_r^s in E_3 the shared registers that can be read by processes in P' have their initial values.*

Proof of Claim 19.3. Suppose, for contradiction, that at time t_r^s in E_3, some shared register R that can be read by a process p' in P' does not have its initial value. By construction, E_3 is identical to E_1 until time t_r^s, and so only processes in P take steps before time t_r^s in E_3. Thus, register R was written by some process p in P by time t_r^s in E_3. Since R is readable by $p' \in P'$ and is written by $p \in P$, R is shared by both p and p'. Thus, there must be a set S_i in L that contains both p and p' – a contradiction to (**). ◀

By Claim 19.3, the shared registers readable by processes in P' have the same value (namely, their initial value) in C_3^0 as in C_2^0. So, C_3^0 also satisfies property (iii). Property (iv) is vacuously true for $i = 0$.

[4] A *step* of $\mathcal{A}$ executed by process p is one of the following: p sending or receiving a message, or p applying a write or a read operation to a shared register in $\mathcal{S}_L$.

[5] The *configuration* of $\mathcal{S}_L$ at time t in execution E consists of the state of each process, the set of messages sent but not yet received, and the value of each shared register in $\mathcal{S}_L$ at time t in E.

For the induction step, for each i such that $1 \leq i \leq \ell$, we consider separately the cases of s^i being a step to send a message, receive a message, write a shared register, and read a shared register. In each case, it is easy to verify that, assuming (inductively) that C_3^{i-1} has properties (i)–(iv), step s^i is applicable to C_3^{i-1}, and the resulting configuration C_3^i has properties (i)–(iv).

To complete the definition of E_3, after time t_r^e we let processes take steps in round-robin fashion. Whenever a process's step is to receive a message, it receives the oldest one sent to it; this ensures that all messages are eventually received. Processes continue taking steps in this fashion according to algorithm $\mathcal{A}$.

Since E_3 is identical to E_1 up to and including time t_w^e, E_3 is indistinguishable from E_1 up to and including time t_w^e to all processes in P. This proves part (a) of the claim.

Note that in E_3 and E_2, the processes in P': (a) take no steps before time t_r^s, and (b) during the time interval $[t_r^s, t_r^e]$, they execute exactly the same sequence of steps, and go through the same sequence of states. Thus, up to and including time t_r^e, E_3 is indistinguishable from E_2 to all processes in P'. This proves part (b) of the claim.

Finally, every process takes steps as required by the algorithm in E_3, so no process crashes. This proves part (c) of the claim. ◀

By Claim 19.2(a), up to and including time t_w^e, E_3 is indistinguishable from E_1 to the writer $w \in P$. So E_3 contains the write operation that writes $v \neq v_0$ into $\mathbf{R}$, which starts at time t_w^s and ends at time t_w^e. By Claim 19.2(b), up to and including time t_r^e, E_3 is indistinguishable from E_2 to $r \in P'$. So E_3 contains the read operation that returns v_0, which starts at time t_r^s and ends at time t_r^e. Since $t_w^e < t_r^s$, this read operation violates Property 1 of atomic SWMR registers. As there are no process crashes in E_3 (by Claim 19.2(c)), this contradicts the assumption that $\mathcal{A}$ is an implementation of an atomic SWMR register $\mathbf{R}$ that tolerates $t > t_L$ crashes. ◀

Note that the proof of Theorem 19 does not depend on the *type* or *number* of registers shared by the processes in each set S_i of the bag L. So the result of this theorem applies not only to $\mathcal{S}_L$ but also to every m&m system in $\mathcal{M}_L$. In fact, the proof of Theorem 19 does not even depend on the type of objects that are shared by the processes in each set S_i; for example these objects could include queues, stacks, and consensus objects. Hence we have the following stronger result:

▶ **Theorem 20.** *Consider any m&m system $\mathcal{S}$ induced by a bag $L = \{S_1, \ldots, S_m\}$ of subsets of $\Pi = \{p_1, p_2, \ldots p_n\}$, where the processes in each S_i share any number of arbitrary objects among themselves (and only among themselves). If more than t_L processes crash in $\mathcal{S}$, then for some process w in $\mathcal{S}$, there is no algorithm that implements an atomic SWMR register writable by w and readable by **all** processes in $\mathcal{S}$.*

4 Atomic SWMR register implementation in uniform m&m systems

Let $G = (V, E)$ be an undirected graph where $V = \{p_1, p_2, \ldots p_n\}$, i.e., the nodes of G are the processes $p_1, p_2, \ldots p_n$. Let $\mathcal{S}_G$ be the uniform m&m system induced by G. Recall that in $\mathcal{S}_G$, each process p_i and its neighbours in G share some atomic SWMR registers that can be read by (and only by) them.

We now use G to determine the maximum number of process crashes that may occur in $\mathcal{S}_G$ such that it is possible to implement a shared atomic SWMR register readable by *all* processes in $\mathcal{S}_G$. To do so, we first recall the definition of the *square of the graph* G: $G^2 = (V, E')$ where $E' = \{(u, v) \mid (u, v) \in E \text{ or } \exists k \in V \text{ such that } (u, k) \in E \text{ and } (k, v) \in E\}$.

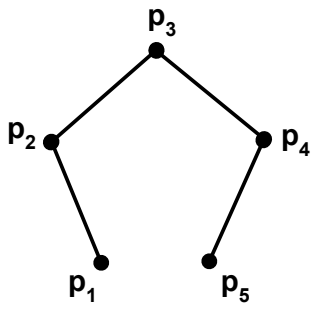


Figure 4 A graph G.

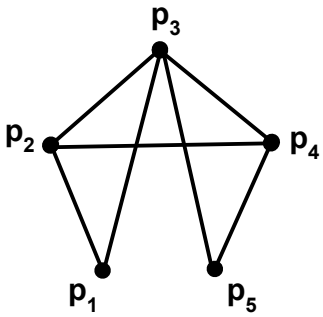


Figure 5 The square of graph G.

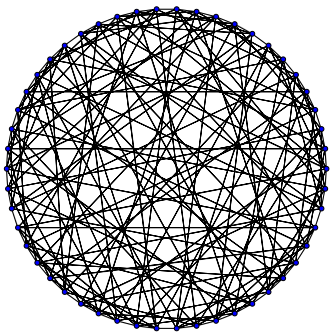


Figure 6 The Hoffman-Singleton graph.

▶ **Definition 21.** *Given an undirected graph $G = (V, E)$ such that $V = \{p_1, p_2, \ldots p_n\}$, t_G is the maximum integer t such that the following condition holds: For all disjoint subsets P and P' of V of size $n - t$ each, some edge in G^2 connects a node in P with a node in P'; i.e., G^2 has an edge (u, v) such that $u \in P$ and $v \in P'$.*

Note that $t_G \geq \lfloor (n-1)/2 \rfloor$. Moreover, in a pure message-passing system (where G and G^2 have no edges) $t_G = \lfloor (n-1)/2 \rfloor$.

In Theorem 22 stated below, we prove that in the uniform m&m system $\mathcal{S}_G$ induced by a graph G, it is possible to implement an atomic SWMR register readable by all processes *if and only if* at most t_G processes may crash in $\mathcal{S}_G$.

For example, consider the graph G in Figure 4 where $V = \{p_1, p_2, p_3, p_4, p_5\}$. Figure 5 shows the corresponding G^2 graph. By the above definition of t_G: (a) $t_G \geq 3$ because for any two disjoint subsets of V of size $5 - 3 = 2$ each, G^2 has an edge that "connects" these two subsets (e.g., for subsets $P = \{p_1, p_2\}$ and $P' = \{p_4, p_5\}$, the edge (p_2, p_4) of G^2 connects a node of P to a node of P'), and (b) $t_G < 4$ because there are two disjoint subsets $\{p_1\}, \{p_5\}$ of size $5 - 4 = 1$ each, such that no edge in G^2 connects p_1 and p_5. So in this example $n = 5$ and $t_G = 3 > \lfloor (5-1)/2 \rfloor = 2$.

Now consider the uniform m&m system $\mathcal{S}_G$ of 5 processes induced by this graph G. In addition to message-passing links, $\mathcal{S}_G$ has 4 pairwise RDMA connections. Since $t_G = 3$, by Theorem 22: (1) we can implement an atomic SWMR register readable by *all* 5 processes of $\mathcal{S}_G$ even if 3 of them (i.e., more than the majority) may crash, and (2) no algorithm can implement such a register in $\mathcal{S}_G$ if more than 3 processes may crash.

As another example, consider a pure message-passing system $\mathcal{S}$ with 50 nodes. In $\mathcal{S}$, one can implement an atomic SWMR register $\mathbf{R}$ (readable by all the processes) only if *at most* 24 process crashes may occur. But if we allow each process of $\mathcal{S}$ to establish 7 pairwise RDMA connections, one can implement $\mathbf{R}$ in a way that tolerates *any number* of process crashes (i.e., $\mathbf{R}$ is wait-free). This is because there is an undirected graph G with $n = 50$ nodes, each with degree 7, such that G^2 has an edge between *every* pair of nodes (G is the well-known Hoffman-Singleton graph [11] shown in Figure 6 [25]); so G has $t_G = n - 1 = 49$, and thus by Theorem 22 it is possible to implement $\mathbf{R}$ in the uniform m&m system $\mathcal{S}_G$ in a way that tolerates up to 49 process crashes. Some simple graph theory arguments show that this is optimal in two ways: (a) one cannot implement a wait-free register $\mathbf{R}$ that is shared by 50 processes with fewer than 7 RDMA connections per process (more precisely, with any such implementation, if a process has fewer than 7 RDMA connections there must be another process with more than 7 RDMA connections), and (b) with at most 7 RDMA connections per process, one cannot implement a wait-free register $\mathbf{R}$ that is shared by more than 50 processes.

▶ **Theorem 22.** *Let $\mathcal{S}_G$ be the uniform m&m system induced by an undirected graph $G = (V, E)$ where $V = \{p_1, p_2, \ldots p_n\}$.*

- *If at most t_G processes crash in $\mathcal{S}_G$, then for every process w in $\mathcal{S}_G$, it is possible to implement an atomic SWMR register writable by w and readable by **all** processes in $\mathcal{S}_G$.*
- *If more than t_G processes crash in $\mathcal{S}_G$, then for some process w in $\mathcal{S}_G$, it is impossible to implement an atomic SWMR register writable by w and readable by **all** processes in $\mathcal{S}_G$.*

Proof. By Definition 6, $\mathcal{S}_G$ is the m&m system $\mathcal{S}_L$ where $L = \{S_1, S_2, \ldots, S_n\}$ such that $S_i = N^+(p_i)$, i.e., for all i, $1 \leq i \leq n$, S_i is the set of neighbours of p_i (including p_i) in the graph G. Recall that t_L is the maximum t such that for all disjoint subsets P and P' of V of size $n - t$ each, some set S_i in L contains both a node in P and a node in P'.

▶ **Claim 22.1.** $t_G = t_L$.

Proof of Claim 22.1. From the definitions of t_G and t_L, it is clear that to prove the claim it suffices to show that for all disjoint subsets P and P' of V of size $n - t$ each, the following holds: some edge in G^2 connects a node in P with a node in P' if and only if some set S_i in L contains both a node in P and a node in P'.

[Only If] Suppose G^2 has an edge (p_i, p_j) such that $p_i \in P$ and $p_j \in P'$; since P and P' are disjoint, p_i and p_j are distinct. By definition of G^2, there are two cases:

1. $(p_i, p_j) \in E$. In this case, $p_j \in N^+(p_i)$ and $p_i \in N^+(p_i)$. So the set $S_i = N^+(p_i)$ in L contains both node $p_i \in P$ and node $p_j \in P'$.
2. There is a node $p_k \in V$ such that $(p_i, p_k) \in E$ and $(p_k, p_j) \in E$. In this case, $p_i \in N^+(p_k)$ and $p_j \in N^+(p_k)$. So the set $S_k = N^+(p_k)$ in L contains both $p_i \in P$ and $p_j \in P'$.

So in both cases, some set S_ℓ in L contains both a node in P and a node in P'.

[If] Suppose set S_k in L contains both a node p_i in P and a node p_j in P'; since P and P' are disjoint, p_i and p_j are distinct. Recall that $S_k = N^+(p_k)$ for node $p_k \in V$.

There are two cases:

1. p_i, p_j and p_k are distinct. In this case, since p_i and p_j are in $S_k = N^+(p_k)$, (p_i, p_k) and (p_k, p_j) are edges of G, i.e., $(p_i, p_k) \in E$ and $(p_k, p_j) \in E$. Thus, by definition of G^2, (p_i, p_j) is an edge of G^2; this edge connects $p_i \in P$ and $p_j \in P'$.
2. $p_k = p_i$ or $p_k = p_j$. Without loss of generality, assume that $p_k = p_i$. Since p_i and p_j are in $N^+(p_k) = N^+(p_i)$, (p_i, p_j) must be an edge of G, i.e., $(p_i, p_j) \in E$. Thus, by definition of G^2, (p_i, p_j) is an edge of G^2; this edge connects $p_i \in P$ and $p_j \in P'$.

So in both cases, some edge in G^2 connects a node in P with a node in P'. ◀

The result now follows immediately from Claim 22.1 and Theorem 8. ◀

5 Concluding remarks

Hybrid systems that combine message passing and shared memory have long been a subject of study in the systems community [3, 5, 6, 7, 16, 17, 20, 23]. To the best of our knowledge, however, such systems have only recently been examined from a theoretical point of view. Aguilera *et al.* gave a rigorous model for hybrid systems, and studied how the combination of message passing and shared memory can be harnessed to improve solutions to certain fundamental problems: In particular, they show that, compared to a pure message-passing system, a hybrid system can improve the fault tolerance of randomized consensus algorithms and reduce the synchrony necessary to elect a leader [1]. A more recent paper by Aguilera *et al.* extends the hybrid model to Byzantine failures, and shows how to improve the inherent trade-off between fault tolerance and performance for consensus, for both Byzantine and

crash failures [2]. The present paper is another contribution to the theoretical study of hybrid systems: whereas the highly cited paper by Attiya *et al.* shows how to implement an atomic SWMR register with optimal fault tolerance in a pure message-passing system [4], here we solve the corresponding problem in hybrid systems. Extending our results to hybrid systems with Byzantine failures is a subject for future research.

References

1. Marcos K. Aguilera, Naama Ben-David, Irina Calciu, Rachid Guerraoui, Erez Petrank, and Sam Toueg. Passing Messages while Sharing Memory. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom, July 23-27, 2018*, pages 51–60, 2018. doi:10.1145/3212734.3212741.
2. Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra Marathe, and Igor Zablotchi. The Impact of RDMA on Agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019.*, pages 409–418, 2019. doi:10.1145/3293611.3331601.
3. Cristiana Amza, Alan L. Cox, Shandya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.
4. Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, January 1995.
5. Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *ACM Symposium on Operating Systems Principles*, pages 29–44, October 2009.
6. John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed Shared Memory Based on Type-specific Memory Coherence. In *ACM Symposium on Principles and Practice of Parallel Programming*, pages 168–176, March 1990.
7. Tudor David, Rachid Guerraoui, and Maysam Yabandeh. Consensus Inside. In *International Middleware Conference*, pages 145–156, December 2014.
8. Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *Symposium on Networked Systems Design and Implementation*, pages 401–414, April 2014.
9. Gen-Z Draft Core Specification—December 2016. URL: http://genzconsortium.org/draft-core-specification-december-2016.
10. Gen-Z DRAM and Persistent Memory Theory of Operation. URL: https://genzconsortium.org/wp-content/uploads/2019/03/Gen-Z-DRAM-PM-Theory-of-Operation-WP.pdf.
11. Alan J. Hoffman and Robert R. Singleton. On Moore Graphs with Diameters 2 and 3. *IBM Journal of Research and Development*, 4(5):497–504, 1960. doi:10.1147/rd.45.0497.
12. InfiniBand. http://www.infinibandta.org/content/pages.php?pg=about_us_infiniband.
13. iWARP. https://en.wikipedia.org/wiki/IWARP.
14. Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA Efficiently for Key-value Services. In *ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 295–306, August 2014.
15. Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *Symposium on Operating Systems Design and Implementation*, pages 185–201, November 2016.
16. Stefanos Kaxiras, David Klaftenegger, Magnus Norgren, Alberto Ros, and Konstantinos Sagonas. Turning Centralized Coherence and Distributed Critical-Section Execution on Their Head: A New Approach for Scalable Distributed Shared Memory. In *IEEE International Symposium on High Performance Distributed Computing*, pages 3–14, June 2015.

17 David Kranz, Kirk Johnson, Anant Agarwal, John Kubiatowicz, and Beng-Hong Lim. Integrating Message-passing and Shared-memory: Early Experience. In *ACM Symposium on Principles and Practice of Parallel Programming*, pages 54–63, 1993.

18 Leslie Lamport. On interprocess communication Part I–II. *Distributed Computing*, 1(2):77–101, May 1986.

19 Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *International Symposium on Computer Architecture*, pages 267–278, June 2009.

20 Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant Software Distributed Shared Memory. In *USENIX Annual Technical Conference*, pages 291–305, July 2015.

21 Marius Poke and Torsten Hoefler. DARE: High-Performance State Machine Replication on RDMA Networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 107–118, New York, NY, USA, 2015. ACM. `doi:10.1145/2749246.2749267`.

22 RDMA over converged ethernet. `https://en.wikipedia.org/wiki/RDMA_over_Converged_Ethernet`.

23 Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A Low Overhead, Software-only Approach for Supporting Fine-grain Shared Memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, October 1996.

24 Shin-Yeh Tsai and Yiying Zhang. LITE kernel RDMA support for datacenter applications. In *ACM Symposium on Operating Systems Principles*, pages 306–324, October 2017.

25 Figure by Uzyel - Own work, CC BY-SA 3.0. `https://commons.wikimedia.org/w/index.php?curid=10378641`.

26 Jian Yang, Joseph Izraelevitz, and Steven Swanson. Orion: A Distributed File System for Non-Volatile Main Memory and RDMA-Capable Networks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 221–234, Boston, MA, February 2019. USENIX Association. URL: `https://www.usenix.org/conference/fast19/presentation/yang`.

Linearizable Replicated State Machines With Lattice Agreement

Xiong Zheng
Electrical and Computer Engineering Department, University of Texas at Austin, USA
zhengxiongtym@utexas.edu

Vijay K. Garg
Electrical and Computer Engineering Department, University of Texas at Austin, USA
garg@ece.utexas.edu

John Kaippallimalil
Wireless Access Laboratories, Huawei, USA
John.Kaippallimalil@huawei.com

Abstract

This paper studies the lattice agreement problem in asynchronous systems and explores its application to building a linearizable replicated state machine (RSM). First, we propose an algorithm to solve the lattice agreement problem in $O(\log f)$ asynchronous rounds, where f is the number of crash failures that the system can tolerate. This is an exponential improvement over the previous best upper bound of $O(f)$. Second, Faleiro et al have shown in [Faleiro et al. PODC, 2012] that combination of conflict-free data types and lattice agreement protocols can be applied to implement a linearizable RSM. They give a Paxos style lattice agreement protocol, which can be adapted to implement a linearizable RSM and guarantee that a command by a client can be learned in at most $O(n)$ message delays, where n is the number of proposers. Later, Xiong et al in [Xiong et al. DISC, 2018] gave a lattice agreement protocol which improves the $O(n)$ message delay guarantee to $O(f)$. However, neither of the protocols is practical for building a linearizable RSM. Thus, in the second part of the paper, we first give an improved protocol based on the one proposed by Xiong et al. Then, we implement a simple linearizable RSM using our improved protocol and compare our implementation with an open source Java implementation of Paxos. Results show that better performance can be obtained by using lattice agreement based protocols to implement a linearizable RSM compared to traditional consensus based protocols.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Lattice Agreement, Generalized Lattice Agreement, Replicated State Machine, Consensus

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.29

Related Version `https://arxiv.org/abs/1810.05871`

Funding This work was partially supported by NSF CSR-1563544,CNS-1812349, WNCG Agreement, and Cullen Trust Professorship.

1 Introduction

Lattice agreement, introduced in [2], to solve the atomic snapshot problem [1] in shared memory, is also an important decision problem in message passing systems. In this problem, n processes start with input values from a lattice and need to decide values which are comparable to each other in spite of f process failures, where n is the number of processes and f is the maximum number of failures in the system.

There are two primary applications of lattice agreement. First, Attiya et al [2] give a $\log n$ rounds algorithm to solve the lattice agreement problem in synchronous message systems and use it as a building block to solve the atomic snapshot problem. Second, Faleiro et al [6]

© Xiong Zheng, Vijay K. Garg, and John Kaippallimalil;
licensed under Creative Commons License CC-BY
23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller; Article No. 29; pp. 29:1–29:16
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

propose the problem of generalized lattice agreement (GLA), which is a generalization of lattice agreement problem for a sequence of inputs, and demonstrate that the combination of conflict-free data types (CRDT) [14, 15] and generalized lattice agreement protocols can be applied to implement a special class of RSM and provide linearizability [8]. We call this special class of state machines as Update-Query (UQ) state machines. The operations of UQ state machines can be classified into two kinds: updates (operations that modify the state) and queries or reads (operations that only return values and do not modify the state). An operation that both modifies the state and returns a value is not supported. In this paper, when we talk about linearizable RSMs, we mean UQ state machines. As shown in [6], to implement a linearizable RSM, we can first design the underlying data structure to be CRDT. This makes all update operations commute. Then, the generalized lattice agreement protocol is invoked for each operation to guarantee linearizability. In this paper, we call a linearizable RSM built by using the combination of CRDT and a GLA protocol as $LaRSM$.

RSM [13] is a popular technique for fault tolerance in a distributed system. Traditional RSMs typically enforce strong consistency among replicas by using a consensus based protocol to order all the requests from the clients. In this approach, each replica executes all the requests in an identical order to ensure that all replicas are at the same state at any given time. The most popular consensus based protocol for building a RSM is Paxos [9, 11]. In Paxos, processes are divided into three different roles: proposer, acceptor and learner. Proposers are responsible for proposing requests from clients to acceptors. Acceptors decide the order of a request and guarantee all learners learn a identical order of requests. When there are multiple proposers in the system, termination is not guaranteed in Paxos. Since the initial proposal of Paxos, many variants have been proposed. FastPaxos [10] reduces the typical three message delays in Paxos to two message delays by allowing clients to directly send commands to acceptors. MultiPaxos [4] is the typical deployment of Paxos in the industrial setting. It assumes that usually there is a stable leader which acts as a proposer, so there is no need for the first phase in the basic Paxos protocol. CheapPaxos [12] extends basic Paxos to reduce the requirement in the number of processors. Even though in the Paxos protocol, there could be multiple proposers, usually only one leader (proposer) is used in practice due to its non-termination problem when there are multiple proposers. The system performance is limited by the resources of the leader. Also, the unbalanced communication pattern limits the utilization of bandwidth available in all of the network links connecting the servers. SPaxos [3] is a Paxos variant which tries to offload the leader by disseminating clients to all replicas. However, the leader is still the only process which can order requests.

Since lattice agreement can be applied to implement a linearizable RSM, if we can solve lattice agreement efficiently, we may not need consensus in some cases. This is promising, since lattice agreement has been shown to be a weaker decision problem than consensus in theory. In synchronous systems, consensus cannot be solved in fewer than $f + 1$ rounds [5], but lattice agreement can be solved in $\log f + 1$ rounds [17]. In asynchronous systems, consensus cannot be solved even with one failure [7], whereas lattice agreement can be solved if a majority of processes is correct [6, 17].

The lattice agreement problem in asynchronous message systems is first studied by Faleiro et al in [6]. They present a Paxos style protocol when a majority of processes are correct. Their algorithm needs $O(n)$ asynchronous round-trips in the worst case. They also propose a protocol for generalized lattice agreement, adapted from their protocol for lattice agreement, which requires $O(n)$ message delays for a value to be learned. Later, a protocol which runs in $O(f)$ asynchronous round-trips was proposed by Xiong et al in [17]. They also give a protocol for generalized lattice agreement which improves the $O(n)$ message delays complexity to $O(f)$. In this work, we improve the upper bound for lattice agreement in asynchronous systems to $O(\log f)$, which is an exponential improvement.

Although [6] has demonstrated that generalized lattice agreement protocol can be applied to implement a linearizable RSM, both the protocols proposed in [6] and [17] are impractical. This is due to the following reason. In both protocols, each process has an accept command which keeps track of all received proposal values. When the protocols are applied to implement a linearizable RSM, this accept command is a set which records all previously proposed commands. When a process rejects a proposal, it has to send back this whole set. Even worse, this set keeps increasing as more commands arrive from clients. In this work, we propose an improved algorithm for the generalized lattice agreement problem, which is specifically designed to make it practical to build a linearizable RSM.

In summary, this paper makes the following contributions:

- We present an algorithm, *AsyncLA*, to solve the lattice agreement in asynchronous system in $O(\log f)$ rounds, where f is the maximum number of crash failures in the system. This bound is an exponential improvement to the previously known best upper bound of $O(f)$ by [17].
- We give an improved algorithm for the generalized lattice agreement protocol based on the one proposed in [17] to make it practical to implement a linearizable RSM.
- We implement a simple linearizable RSM in Java by combining a CRDT map data structure and our improved generalized lattice agreement algorithm. We demonstrate its performance by comparing with SPaxos. Our experiments show that LaRSM achieves around 1.3x times throughput than SPaxos and a lower operation latency in normal case.

2 System Model and Problem Definitions

2.1 System Model

We consider a distributed message passing system with n processes, $p_1, \ldots, p_n$, in a completely connected topology. We only consider asynchronous systems, which means that there is no upper bound on the time for a message to reach its destination. The model assumes that processes may have crash failures but no Byzantine failures. The model parameter f denotes the maximum number of processes that may crash in a run. We do not assume that the underlying communication system is reliable.

2.2 Lattice Agreement

In the lattice agreement problem, given a join semi-lattice $(X, \leq, \sqcup)$ with $\leq$ as the partial order and $\sqcup$ as the join operation, each process p_i proposes a value x_i in X and must decide on some output y_i also in X. An algorithm solves the lattice agreement problem if the following properties are satisfied:

Downward-Validity: For all correct processes $i \in [1..n]$, $x_i \leq y_i$.

Upward-Validity: For all correct processes $i \in [1..n]$, $y_i \leq \sqcup\{x_1, ..., x_n\}$.

Comparability: For any two correct $i \in [1..n]$ and $j \in [1..n]$, either $y_i \leq y_j$ or $y_j \leq y_i$.

2.3 Generalized Lattice Agreement

In the generalized lattice agreement problem [6], each process may receive a possibly infinite sequence of values belong to a lattice at any point of time. Let x_i^p denote the ith value received by process p. The aim is for each process p to learn a sequence of output values y_j^p which satisfies the following conditions:

Validity: Any learned value y_j^p is a join of some subset of received input values.

Stability: The value learned by any correct p is non-decreasing: $j < k \implies y_j^p \leq y_k^p$.

Comparability: Any two values y_j^p and y_k^q learned by any two correct processes p and q are comparable.

Liveness: Every value x_i^p received by a correct process p is eventually included in some learned value y_k^q of every correct process q: i.e, $x_i^p \leq y_k^q$.

3 Asynchronous Lattice Agreement

In this section, we give an algorithm to solve the lattice agreement problem in asynchronous systems which only needs $O(\log f)$ asynchronous rounds. The proposed algorithm is inspired by the algorithm for synchronous setting in [17]. The basic idea is to apply a *Classifier* procedure, which is associated with a specific threshold value, to divide processes into *master* and *slave* groups and ensure that any process in the *master* group have values great than or equal to any process in the *slave* group. Then, by recursively applying a *Classifier* procedure within each subgroup, eventually all processes have comparable values. Equivalently, we can think of the above recursive procedure as letting all processes traverse through a virtual binary *Classifier* tree. Each node of this tree has a *Classifier* procedure with a specific threshold value. When traversing through a node in this tree, the processes will invoke the *Classifier* procedure at this node. Some processes will be classified as *master* and go to the right child, and others will be classified as *slave* and go to the left child. The threshold value associated with the *Classifier* procedure is used to decide which processes should be classified as *master* and which should be classified as *slave*. By carefully setting the threshold value for each *Classifier* procedure in the virtual tree, we can make sure all processes eventually have comparable values.

The main difficulty of the above recursive procedure lies in constructing a *Classifier* procedure to divide a group of processes into two subgroups such that processes in one subgroup have values greater than or equal to processes in the other subgroup. In synchronous systems, [17] gives a very simple procedure. First, each process sends its value to all processes within the same group. Then, it takes join of all values received from processes in the same group. If the join is greater than the threshold value, it is classified as *master* and updates its value to be the join. Otherwise, it is classified as *slave* and keeps its value unchanged. We can easily see that the value of each *slave* process must be received by each *master* process, since the system is synchronous. Thus, each *master* process has value greater than or equal to each *slave* process. In asynchronous systems, however, it is not straightforward to design such a *Classifier* procedure, since we cannot guarantee that the value of a *slave* process is received by each *master* process. Our primary idea for such a *Classifier* procedure in asynchronous systems is as follows. In an asynchronous system, at each round, each process can only wait for $n - f$. If we can guarantee that (1) the value of a *slave* process is stored in at least $n - f$ processes at some point, and (2) each *master* process reads from at least $n - f$ processes, then the value of each *slave* process must be known by each *master* process. This claim holds since any two group of $n - f$ processes have at least one process in common, if we assume $f < \frac{n}{2}$.

The virtual *Classifier* tree is built based on the knowledge of the height of the input lattice, which is unknown. Thus, instead of directly agreeing on the input value lattice, we first agree on a view lattice, which has a known maximum height.

We associate each process p_i with a view v_i, which is an array composed of n entries. Each entry of the view corresponds to the input value of each process known by p_i. Initially, $v_i[i] = x_i$ and $\forall j \neq i, v_i[j] = \perp$, where x_i is the input value of p_i. We say $\perp$ is smaller that any input value. For any two views v and u, we say v *dominates* u, if for all i, $v[i] \geq u[i]$.

Consider the lattice formed by the initial views of all processes with the order defined by the domination relation, i.e, $v \leq u$ iff u dominates v. We call this lattice the view lattice. This view lattice has its smallest element (or bottom) equal to $[\bot, ..., \bot]$ and the top element equal to $[x_1, ..., x_n]$. The height of the view lattice is n (the length of the longest chain). We say v and u are comparable if either $v \leq u$ or $u \leq v$. The join of any two views is defined as the component-wise maximum. The height of a view v, denoted as $h(v)$, is defined as the number of components which are not $\bot$, i.e, the number of processes whose values are contained in this view. Since the ith entry of any view is either the input value of p_i or $\bot$, if a view $v \leq u$, then view u contains all input values contained in view v. That is, in the original input lattice, we have $\sqcup\{v[i] : i \in [1..n]\} \leq \sqcup\{u[i] : i \in [1..n]\}$ if $v \leq u$. Thus, if all correct processes can output comparable views from the view lattice, they can output comparable values from the input value lattice by taking join of all values contained in its output view. Therefore, in our algorithm, instead of directly working on the input value lattice, we apply the *Classifier* technique on the view lattice. The *Classifier* procedure is shown in Fig. 1. The main algorithm, *AsyncLA*, is shown in Fig. 2.

3.1 The Classifier Procedure

The *Classifier* procedure has three input parameters: the input view, the threshold value, and the round number. Each process keeps a label. Whenever a process invokes the *Classifier* procedure, it passes its current view, its label and the current round number as the parameters. We say any two processes p_i and p_j are in the same group at a certain round if they invoke the *Classifier* procedure with the same threshold value, i.e, they have the same label. When processes are classified into different subgroups, they update their labels accordingly (to be explained later). Since processes pass their labels as the threshold value of the *Classifier* procedure, we use label or threshold value interchangeably henceforth. Details of the *Classifier* procedure for p_i at round r are shown as below:

Line 0: p_i set its $acceptVal_r$ to be empty. This $acceptVal_r$ set is used to record all the $< view, label >$ pairs received from all processes at round r via *write* or *read* messages. Note that this $acceptVal_r$ also includes $< view, label >$ pairs received from processes that are not in the same group as p_i.

Line 1-2: p_i sends a *write* message with its current view v and the threshold value (current label) k to all processes and waits for $n - f$ *write_ack*s. This step is to ensure that the value and label of p_i is in the $acceptVal_r$ set of $n - f$ processes.

Line 3-5: p_i sends a *read* message with its current round number r to all processes and waits for $n - f$ *read_ack*s. It collects all received views associated with the same label k in a set U, i.e, collects all views from processes within the same group. It may seem that *lines* 3-5 perform the same functionality as *lines* 1-2 and there is no need to have both. However, this part is actually the key of the *Classifier* procedure. The reason will be clear in the correctness proof section.

Line 6-14: p_i performs classification based on the views received from processes in the same group. Let w be the join of all received views in U. If the height of w is greater than k, then p_i sends a *write* message with w, k and r to all and waits for $n - f$ *write_ack*s with round number r. Then in *line* 10-12, it takes the join of w and all the views contained in the *write_ack*s from the same group, denoted as w'. It returns $(w', master)$ as output of the *Classifier* procedure in which *master* indicates its classified into *master* group in the next round. Otherwise, it returns its own input view v and *slave*.

When p_i receives a *write* message for round r_j from p_j, it includes the $< view, label >$ pair contained in the message into its $acceptVal_{r_j}$ set and sends a *write_ack* message containing the current $acceptVal_{r_j}$ back. When p_i receives a *read* message for round r_j from

p_j, it sends a *read_ack* message containing its current $acceptVal_{r_j}$ back. Basically, the *write* message is used to ensure at least $n - f$ processes know the current view and label of a process and the *read* message is used to retrieve the knowledge of at least $n - f$ processes.

Note that when a process which is invoking the *Classifier* at round r receives a *write* or *read* message with a round number $r' > r$, it buffers this message and delivers it when it reaches round r'.

```
Classifier(v, k, r):
v: input view
k: threshold value
r: round number

0: acceptVal_r := ∅ // set of <view, threshold> pairs.

/* write */
1: Send write(v, k, r) to all
2: wait for n − f write_ack(−, −, r)

/* read */
3: Send read(r) to all
4: wait for n − f read_ack(−, −, r)
5: Let U be views contained in received acks with label equals k
/* Classification */
6: Let w := ⊔{u : u ∈ U}
7: if h(w) > k  /* height of w is greater than its label */
8:     Send write(w, k, r) to all
9:     wait for n − f write_ack(−, −, r)
10:    Let U' be views contained in received acks with label equals k  /* views received in the same group */
11:    Let w' := w ⊔ {u : u ∈ U'}
12:    return (w', master)
13: else
14:    return (v, slave)

Upon receiving write(v_j, k_j, r_j) from p_j
    acceptVal_{r_j} := acceptVal_{r_j} ∪ < v_j, k_j >
    Send write_ack(acceptVal_{r_j}, r_j) to p_j

Upon receiving read(r_j) from p_j
    Send read_ack(acceptVal_{r_j}, r_j) to p_j
```

Figure 1 *The Classifier Procedure.*

3.2 Algorithm *AsyncLA*

Now let us look at the main algorithm, *AsyncLA*. The basic idea of *AsyncLA* is to let all processes recursively invoke the *Classifier* procedure with a carefully set threshold value. Let y_i denote the output value of p_i. Let v_i^r denote its view at the beginning of round r. The algorithm for p_i proceeds in asynchronous rounds. The algorithm runs in $\log f + 1$ rounds.

At round 0, all processes exchange their views. The purpose of round 0 is to allow us to construct the virtual *Classifier* tree with height equal to $\log f$. In such case, the recursive invocation of the *Classifier* procedure terminate in $\log f$ rounds. The reason is as follows. After round 0, the view of each correct process must have height at least $n - f$ in the view lattice. Since the height of the view lattice is n, the join-closed subset that includes all current views after round 0 (which is also a lattice) has height at most f. Then we can construct a the binary *Classifier* tree with height equals to $\log f$ by setting the threshold value of the root *Classifier* to be $\frac{(n-f)+n}{2} = n - \frac{f}{2}$ in the virtual tree. We say the root *Classifier* is at level 1. For any node at level r of the tree with threshold value k, we set the threshold value of its right child to be $k + \frac{f}{2^{r+1}}$ and the threshold value of its left child to be $k - \frac{f}{2^{r+1}}$. Thus, we can easily see that the height of the tree is $\log f$. Note that the initial label of each process is $n - \frac{f}{2}$, which means all processes are at the root of the tree.

From round 1 to $\log f$, each process simply traverses the virtual *Classifier* tree. At round r, each process is at level r of the tree. For process p_i, it invokes the *Classifier* procedure with its current view v_i^r, current label l_i and r as parameters. Based on the output of the

Classifier procedure, p_i adjust its label to be the threshold value of the *Classifier* procedure it will invoke at next round. If it is classified as master, then it increases its label by $\frac{f}{2^{r+1}}$, i.e, it goes to the right subtree of the virtual *Classifier* tree. Otherwise, it reduces its label by $\frac{f}{2^{r+1}}$, i.e, it goes to the left subtree of the virtual *Classifier* tree. At the end of round $\log f$, p_i outputs the join of all values contained in its current view as its decision value.

```
AsyncLA(x_i) for p_i:
x_i: input value
y_i: output value
l_i := n - f/2 // label of p_i
v_i^r: the view of p_i at the beginning of round
r, an array of size n.
Initially, v_i^0[i] = x_i ∧ v_i^0[j] = ⊥, ∀j ≠ i

/* Round 0 */
Send value(v_i^0, 0) to all
wait for n - f messages of form value(-, 0)
Let U denote the set of all received values
/* Round 1 to log f */
v_i^1 := ⊔{u | u ∈ U}
for r := 1 to log f
    (v_i^{r+1}, class) := Classifier(v_i^r, l_i, r)
    if class = master
        l_i := l_i + f/2^{r+1}
    else
        l_i := l_i - f/2^{r+1}
end for
Let V_i := v_i^{log f+1}
y_i := ⊔{V_i[j] : j ∈ [1..n]}
```

Figure 2 Algorithm *AsyncLA*.

3.3 Proof of Correctness

Let us prove the correctness of *AsyncLA*. Let w_i^r be the value of w at *line* 6 of the *Classifier* procedure at round r. Let G be a group of processes at round r. Recall that a group G at round r is a set of processes which have the same label at round r. The label of a group is the label of the processes in this group. Let $M(G)$ and $S(G)$ be the group of processes which are classified as *master* and *slave*, respectively, when they run the *Classifier* procedure in group G. Recall that $h(v)$ denote the height of view v in the view lattice. The following lemma presents the key properties of the *Classifier* procedure, with detailed proof given in the full paper [16].

▶ **Lemma 1.** *Let G be a group at round r with label k. Let L and R be two nonnegative integers such that $L \leq k \leq R$. If $L < h(v_i^r) \leq R$ for every process $i \in G$, and $h(\sqcup\{v_i^r : i \in G\}) \leq R$, then*

(p1) *for each process $i \in M(G)$, $k < h(v_i^{r+1}) \leq R$*
(p2) *for each process $i \in S(G)$, $L < h(v_i^{r+1}) \leq k$*
(p3) $h(\sqcup\{v_i^{r+1} : i \in M(G)\}) \leq R$
(p4) $h(\sqcup\{v_i^{r+1} : i \in S(G)\}) \leq k$, *and*
(p5) *for each process $i \in M(G)$, $v_i^{r+1} \geq \sqcup\{v_i^{r+1} : i \in S(G)\}$*

Once we have the above lemma, the correctness proof for *AsyncLA* follows in a similar fashion as the synchronous algorithm in [17]. We only give the primary lemmas and put the detailed proof in the full paper [16].

▶ **Lemma 2.** *Let G be a group of processes at round r with label k. Then*

(1) *for each process $i \in G$, $k - \frac{f}{2^r} < h(v_i^r) \leq k + \frac{f}{2^r}$*
(2) $h(\sqcup\{v_i^r : i \in G\}) \leq k + \frac{f}{2^r}$

▶ **Lemma 3.** *Let i and j be two processes that are within the same group G at the end of round $r = \log f$. Then v_i^{r+1} and v_j^{r+1} are equal.*

Proof. (Sketch of Proof) Intuitively, after round 0, the join-closed subset that includes all current views after round 0 (which is also a lattice) has height at most f. We know that the height interval of values in a group are shrinking by a factor of 2, from Lemma 2. Thus, after $\log f$ rounds, any two processes in a same group must have a same value. ◀

▶ **Lemma 4.** *Let process i decides on y_i. Let G be a group at round r such that $i \in S(G)$, then $y_i \leq \sqcup\{v_i^{r+1} : i \in S(G)\}$.*

Proof. Immediate from $(p2)$ and $(p4)$ of Lemma 1. ◀

Since the value of a process is non-decreasing at each round, from $(p5)$ of Lemma 1 and Lemma 4, we have that once two processes are classified into two subgroups, their values must be comparable. We immediately have the following lemma.

▶ **Lemma 5.** *Let i and j be any two processes in two different groups G_i and G_j at the end of round $\log f$, then y_i is comparable with y_j.*

▶ **Theorem 6.** *Algorithm* AsyncLA *solves the lattice agreement problem in $O(\log f)$ round-trips when at least a majority of processes are correct.*

Proof. *Down-Validity* holds since the value held by each process is non-decreasing. *Upward-Validity* follows because each learned value must be the join of a subset of all initial values which is at most $\sqcup\{x_1, ..., x_n\}$. For *Comparability*, from Lemma 3, we know that any two processes which are in the same group at the end of *AsyncLA*, they must have equals values. For any two processes which are in two different groups, from Lemma 5 we know they must have comparable values. ◀

3.4 Complexity Analysis

Each invocation of the *Classifier* procedure takes at most three round-trips. Therefore, $\log f$ invocation of *Classfier* results in at most $3 * \log f$ round-trips. Thus, the total time complexity is $3 * \log f + 1$ round-trips. For message complexity, each process sends out at most 3 *write* and *read* messages and at most $3 * n$ *write_ack* and *read_ack* messages. Therefore, the message complexity for each process is $O(n * \log f)$.

4 Improved Generalized Lattice Agreement Protocol for RSM

In this part, we give optimizations for the generalized lattice agreement protocol proposed in [17] (referred as GLA) to implement a linearizable RSM. Inside GLA, a $f + 1$ round-trips asynchronous lattice agreement protocol proposed in the same paper is embedded. We do not change their $f + 1$ round-trips asynchronous protocol to our $O(\log f)$ protocol for reasons as follows. First, our primary goal in this section is not to give a new protocol for the generalized lattice agreement problem, yet to make it practical to implement. The optimizations we make in this section do not involve the lattice agreement protocol part. Keeping the simple $f + 1$ round-trips algorithm would make things easier. Second, in practice, the $f + 1$ round-trips algorithm is favorable compared to the $O(\log f)$ algorithm due to its smaller constant.

The optimized protocol, GLA_Δ, is shown in Fig. 3 with the two main changes marked using Δ. Note that although we only have two primary changes compared to GLA, we claim those changes are the key for its applicability in building a linearizable RSM. The basic idea of GLA_Δ is to invoke a separate lattice agreement instance for a set of concurrent commands. To ensure *Comparability* and *Stability* of generalized lattice agreement, we associate each

lattice agreement instance with a distinct sequence number and ensure that lattice agreement instance with higher sequence number only starts running after the instance with smaller sequence number has completed. In this way, we guarantee that any value learned by higher sequence lattice agreement instance is greater than or equal to the value learned by lower sequence lattice agreement instance. Along with the *Comparability* property of each lattice agreement instance, we can obtain *Comparability* and *Stability* for the generalized lattice agreement problem. Details are given in correctness proof section.

In GLA_Δ, the sequence number assigned for lattice agreement instances starts from 0 and increase by 1 when a new lattice agreement instance is created. Each process keeps an integer s, which is the next available sequence number. Since different processes might be executing lattice agreement instance with different sequence number, *maxSeq* is used to record the largest sequence number known by a process. *buffVal* stores all the commands received which need to be learned. *LV* denotes the mapping from a sequence number to its corresponding learned command set. *acceptVal* stores all commands received from all processes via proposal messages.

```
GLA_Δ for p_i:
s := 0 // sequence number
maxSeq := -1 // largest sequence number seen
buffVal := ⊥ // commands buffer
LV := ⊥ // map from seq to learned commands set
acceptVal := ⊥ // current accepted commands set
active := false //proposing status

Procedure ReceiveValue(v):
    buffVal := buffVal ⊔ v

on receiving prop(v_j, r, s') from p_j:
    if s' < s
        buffVal := buffVal ⊔ v_j        Δ_2
        Send ACK("decide", LV[s'], r, s')
        return
    maxSeq := max{s', maxSeq}
    wait until s' = s
    if acceptVal ⊆ v_j
        Send ACK("accept", −, r, s')
        acceptVal := v_j
    else
        Send ACK("reject",acceptVal, r, s')

Procedure Agree():
guard: (active = false) ∧ (buffVal ≠ ⊥ ∨ maxSeq ≥ s)
    active := true
    acceptVal := buffVal ⊔ acceptVal
    buffVal := ⊥
    /* Lattice Agreement with sequence number s */
    for r := 1 to f + 1
        val := acceptVal
        Send prop(val, r, s) to all
        wait for n − f ACK(−, −, r, s)
        Let V be values in reject ACKs
        Let D be values in decide ACKs
        Let tal be the number of accept ACKs
        if |D| > 0
            val := ⊔{d | d ∈ D} ; break
        else if tal > n/2
            break
        else
            acceptVal := acceptVal ⊔ {v | v ∈ V}
        end
    LV[s] := val
    acceptVal := acceptVal - LV[s − 1]  Δ_1
    s := s + 1
    active := false
```

Figure 3 Algorithm GLA_Δ.

When a process receives a command v from a client, it invokes *ReceiveValue(v)* to include v into its buffer (*buffVal*).

Each process invokes the *Agree()* procedure to start a new lattice agreement instance. This procedure is automatically executed when the guard condition is satisfied. Inside the *Agree()* procedure, a process first updates its *acceptVal* to be the join of current *acceptVal* and *buffVal*. Then, it starts a lattice agreement instance with the next available sequence number. The lattice agreement instance runs the $f + 1$ round-trips protocol in [17]. At each round of the lattice agreement, a process sends its current *acceptVal* to all and waits for $n - f$ *ACK*s. If it receives any *decide ACK*, it decides on the join of all *decide* values. If it receives a majority of *accept ACK*s, it decides on its current value. Otherwise, it updates its *acceptVal* to be the join of all received values and starts next round. When a process receives a proposal from some other process, if the proposal is associated with a smaller sequence number, then it sends *decide ACK*s back with its decided value for that sequence number and includes the received value into its own buffer set. Otherwise, it waits until its current sequence number to reach the sequence number associated with the proposal. Then, it checks whether the proposed value contains its current *acceptVal*. If true, the process sends back a *accept ACK*. Otherwise, it sends back a *reject ACK* along with its current *acceptVal*. When a process completes lattice agreement for sequence number s, it stores learned values in *LV[s]* and removes all learned values for sequence number $s - 1$.

Now we explain the our proposed improvements in detail.

4.1 Truncate the Accept Command Set

Let us first look at the challenges of directly applying the GLA protocol in [17] or the one in [6] to implement a linearizable RSM. In a RSM, each input value is a command from a client. Thus, the input lattice is a finite boolean lattice formed by the set of all possible commands. The order in this lattice is defined by the set inclusion, and the join is defined as the union of two sets. This boolean input lattice poses a challenge for both the algorithms in [6] and [17]. In these algorithms, for each process there is an accept command set (*acceptVal*), which stores the join of whatever value the process has accepted. Now since the join is defined as union in the RSM setting, this set keeps increasing. For example, in the original algorithm given in [17], it does not include the line marked as Δ_1. Suppose, we have three processes p_1, p_2 and p_3 which handles commands from clients. Suppose p_1, p_2 and p_3 first receive commands $\{a\}$, $\{b\}$ and $\{c\}$, respectively. They start the lattice agreement instance with the sequence number 0 and learn $\{a\}$, $\{a, b\}$ and $\{a, b, c\}$ respectively for the sequence number 0. After that, p_1, p_2 and p_3 receive $\{d\}$, $\{e\}$, and $\{f\}$ as input, respectively. Now, they start a lattice agreement instance with the sequence number 1. In order to ensure *Comparability* and *Stability* of GLA, the accept command set for sequence number 1 have to include the largest learned value of sequence 0, which is $\{a, b, c\}$, although each process only proposes a single command. Therefore, the accept command set keeps increasing. This problem makes applying lattice agreement to implement a linearizable RSM impractical.

To tackle the above problem, we need to truncate the accept command set. A naive way is to remove all learned commands in the accept command set when proposing for the next available sequence number. This way does not work. Suppose we have two processes: p_1, p_2 and p_3. They propose $\{a\}$, $\{b\}$ and $\{c\}$, respectively for sequence number 0. After execution of lattice agreement for sequence number 0, suppose p_1, p_2 and p_3 both have learned value set and accept command set to be $\{a\}$, $\{a, b, c\}$, and $\{a, b, c\}$, respectively. It is easy to verify this case is possible for an execution of lattice agreement. When completing sequence number 0, all processes remove learned value set for sequence number 0 from their accept command set. Thus, the accept command set of all the three processes becomes to be empty. Now, suppose p_1, p_2 and p_3 start to propose for sequence number 1 with new

commands $\{d\}$, $\{e\}$ and $\{f\}$. Since the accept command sets of p_2 and p_3 do not contain value $\{b\}$ and $\{c\}$, p_1 will never be able to learn $\{b\}$ and $\{c\}$. Thus, learned command set of p_1 for sequence 1 and the learned command set of p_2 and p_3 for sequence 0 are incomparable. Instead of removing all learned commands from the accept command set, we propose to remove all learned commands for the sequence numbers smaller than the largest learned sequence number from the accepted command set. In order to achieve this, the line marked by Δ_1 in the pseudocode is added, compared to the original algorithm in [17]. In this line, after a process has learned a value set for sequence number s, it removes the learned value set corresponding to sequence number $s-1$ from its accept command set.

4.2 Remove Forwarding

In both the algorithms of [6] and [17], a process has to forward all commands it receives to all other processes to ensure liveness. This forwarding results in load that is multiplied many fold, since many processes may propose the same request. In [17], this forwarding is to ensure that the commands proposed by slow processes can also be learned. However, for the fast processes, there is no need to forward their requests to others because they can learn requests quickly. Therefore, instead of forwarding every request to all servers, we require that when a process receives some proposal with a smaller sequence number than its current sequence number, it sends back a *decide* message and also includes the received proposal value into its buffer set. These values will be proposed by the server in its next sequence number. In this way, only when a process is slow, its value will be proposed by the fast processes. This change is shown as addition of the line marked by Δ_2.

4.3 Proof of Correctness

Let us prove the correctness of GLA_Δ. Although we only have two primary changes compared to the algorithm in [17], the correctness proof is quite different due to the modification marked by Δ_1. Let $LV_p[s]$ denote the learned value of process p for sequence number s. Let $LearnedVal_s^p$ denote all the learned values of process p after completing lattice agreement for sequence number s. Thus, $LearnedVal_s^p = \sqcup\{LV_p[t] : t \in [0...s]\}$. Due to the page limit, we put the proof for Lemma 7-9 in the full paper [16].

▶ **Lemma 7.** *For any sequence number s, $LV_p[s]$ is comparable with $LV_q[s]$ for any two processes p and q.*

▶ **Lemma 8.** *For any sequence number s, $LearnedVal_p^s \subseteq LearnedVal_q^{s+1}$ for any two correct processes p and q.*

▶ **Lemma 9.** *For any sequence number s and s', $LearnedVal_p^s$ and $LearnedVal_q^{s'}$ are comparable for any two correct processes p and q.*

▶ **Theorem 10.** *Algorithm GLA_Δ solves the generalized lattice agreement problem when a majority of processes is correct.*

Proof. *Validity* holds since any learned value is the join of a subset of values received. *Stability* follows from Lemma 8. *Comparability* follows from Lemma 9. *Liveness* follows from the termination of lattice agreement. ◀

5 LaRSM vs Paxos

In this section, we compare LaRSM and Paxos from both theoretical and engineering perspective. Table 1 shows the theoretical perspective. The main difference between Paxos and LaRSM lies in their termination guarantee. In the worst case, Paxos may not terminate (∞ message delays), though very unlikely. Whereas, LaRSM always guarantee termination in at most $O(\log f)$ message delays. This difference is because Paxos is consensus based and LaRSM is lattice agreement based. In the best case, both Paxos and LaRSM need three message delays. One limitation of LaRSM is that it is only applicable to UQ state machines.

For engineering perspective, Paxos is typically deployed with only one single proposer (leader) due to its non-termination. Only the leader can handle handle requests from clients. Thus, in a typical deployment the leader becomes the bottleneck, i.e, the throughput of the system is limited by the leader's resources. Besides, the unbalanced communication pattern limits the utilization of bandwidth available in all of the network links connecting the servers. In LaRSM, however, there could be multiple proposers since termination is guaranteed, which can simultaneously handle requests from clients and may yield better throughput. In the failure case, a new leader needs to be elected in Paxos and there could be multiple leaders in the system. During this time, Paxos generally takes longer to terminate because of conflicting proposals. However, a failure of a replica in LaRSM has limited impact on the whole system. This is because other replicas can still handle requests from clients as long as less than a majority of replicas has failed.

Table 1 Paxos vs LaRSM.

Properties	**Paxos**	**LaRSM**
Consistency	Linearizability	Linearizability
Underlying Protocol	Consensus	Lattice Agreement
Best Case #Message Delays	3	3
Worse Case #Message Delays	∞	$O(\log f)$
Applicable to All Sate Machines	Yes	Only UQ State Machines

6 Evaluation

In this section, we evaluate the performance of LaRSM and compare with SPaxos. To implement LaRSM, we also propose some practical optimizations for the procedure proposed in [6] to implement a RSM by combining CRDT and a generalized lattice agreement protocol, which can be found in the full paper [16].

Although the lattice agreement protocol proposed in this paper has round complexity of $O(\log f)$, it has a large constant, which is only advantageous when the number of replicas is large. In practical cases, the number of replicas is usually small, often 3 to 5 nodes. Thus, we adopt the lattice agreement protocol from [17] which runs in $f + 1$ asynchronous round-trips in our implementation. In order to evaluate LaRSM, we implemented a simple RSM which stores a Java hash map data structure. We implement the hash map date structure to be a CRDT by assigning a timestamp to each update operation and maintain the last writer wins semantics. We measure the performance of SPaxos and our implementation in the following three perspectives: performance in the normal case (no crash failure), performance in failure case, and performance under different work loads.

All experiments are performed on Amazon's EC2 micro instances, which have variable ECUs (EC2 Compute Unit), 1 vCPUs, 1 GBytes memory, and low to moderate network performance. All servers run Ubuntu Server 16.04 LTS (HVM) and the socket buffer sizes are equal to 16 MBytes. All experiments are performed in a LAN environment with all processes distributed among the following three availability zones: US-West-2a, US-West-2b and US-West-2c.

The keys and values of the map are string type. We limit the range of keys to be within 0 to 1000. Two operations are supported: update and get. The update operation changes the value of a specific key. The get operation returns the value for a specific key. A client execute one request per time and starts executing next request when it completes the current one. The request size is 20 bytes. For each request, the server returns a response to indicates its completion. In order to compare with SPaxos, we set its crash model to be CrashStop. In this model, SPaxos would not write records into stable storage. In SPaxos, batching and pipelining are implemented to increase the performance of Paxos. There are some parameters related to those two modules: the batch size, batch waiting timeout and the window size. The batch size controls how many requests the batcher needs to wait before starting proposing for a batch. The batch waiting timeout controls the maximum time the batch can wait for a batch. The window size is the maximum number of parallel proposals ongoing. We set the batch size to be 64KB, which is the largest message size in a typical system. We set the batch timeout according to the number of clients from 0 to 10 at most. The window size is set to 2 because we found that increasing the window size further does not improve the performance in our evaluation.

6.1 Performance in Normal Case

In this experiment, we build a RSM system with three instances. We measure the throughput of the system and latency of operations while increasing the number of clients. The load from the clients are composed of 50% writes and 50% reads. The left part of Fig. 4 shows the throughput of SPaxos and LaRSM. The throughput is measured by the number of requests handled per second by the system. The latency is the average time in milliseconds taken by the clients to complete execution of a request. We can see from Fig. 4, as we increase the number of clients, the throughput of both SPaxos and LaRSM increases until there are around 1000 clients. At that point, the system reaches its maximum handling capability. If we further increase the clients number, the throughput of both LaRSM and SPaxos does not change in a certain range and begins to decrease. This is because both systems do not limit the number of connections from the client side. A large number of clients connection results in large burden on IO, decreasing the system performance. Comparing SPaxos and LaRSM, we can see that LaRSM always has better throughput than SPaxos.

The right part of Fig. 4 shows the latency of LaRSM and SPaxos. In both LaRSM and SPaxos, read and write perform the same procedure, thus their latency should be similar. So, in our evaluation, we just use operation latency. From Fig. 4, we find that operation latency of LaRSM keeps increasing. As we increase the number of clients, the latency of SPaxos decreases first up to some point and then begins to increase. This performance is because the latency is the average response time of all clients and SPaxos has a batching module which batches multiple requests from different clients to propose in a single proposal. Thus, initially when there are very few clients, they can only propose a small number of requests in a single proposal, which makes the latency relatively higher. While the number of clients increases, more requests can be proposed in one single batch, thus the average latency for one client decreases. If the number of clients increases further, the handling capability limit

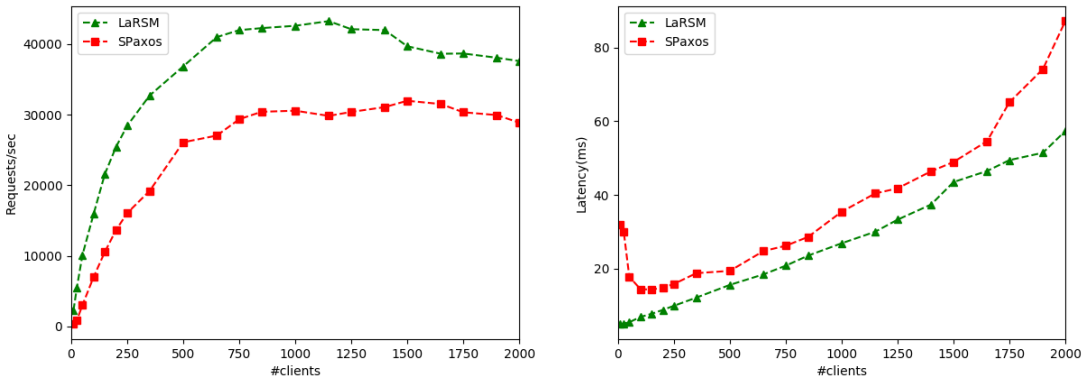


Figure 4 Throughput and latency of LaRSM and SPaxos with increasing number of clients.

of the system increases the operation latency. Comparing SPaxos and LaRSM, we find that the latency of LaRSM is always around 5ms smaller.

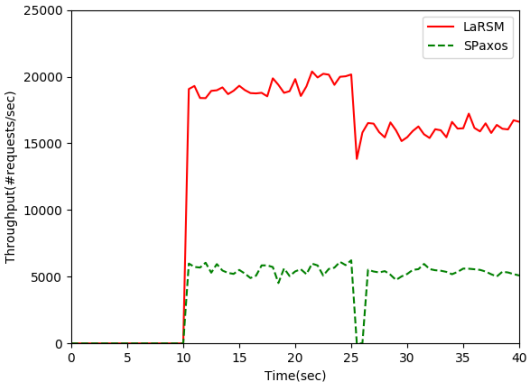


Figure 5 Throughput in Case of Failure.

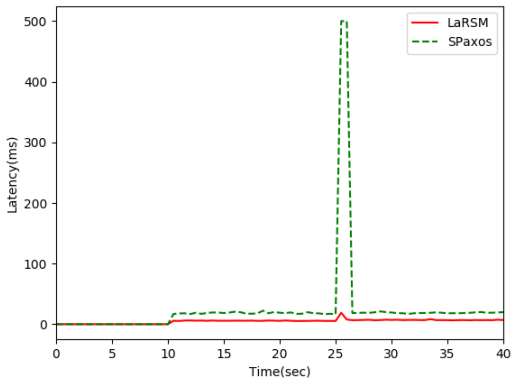


Figure 6 Latency in Case of Failure.

6.2 Performance in Failure Case

In this section, we evaluate the performance of both LaRSM and SPaxos in the case of failure. In this experiment, the RSM system is composed of five replicas. There are 100 clients that keep issuing requests to the system. In LaRSM, since all replicas perform the same role and can handle requests from the clients concurrently. Thus, for loading balancing, each client randomly selects a replica to connect. Each client has a timeout, unlike SPaxos, this timeout is typically small. Timeout on an operation does not necessarily mean failure of the connected replica. It might also be due to an overload of the replica. In this case, the client randomly chooses another replica to connect. However, in SPaxos, the timeout set for a client is usually used to suspect the leader. That is, when an operation times out, most likely the leader has failed. Thus, the timeout in SPaxos is typically large.

We run the simulation for 40 seconds. The first 10 seconds is for the system to warm up, so we do not record the throughput and latency data. A crash failure is triggered at 25th second after the start of the system. For LaRSM, we randomly shut down one replica since

all replicas are performing the same role. For SPaxos, we shut down the leader, since crash of a follower does not have much impact on the system. Figure 5 shows the throughput of both LaRSM and SPaxos. Figure 6 shows the latency change. From Fig. 5 and Fig. 6, for LaRSM we can see that when the failure occurs, the throughput drops sharply from around 20K requests/sec to around 15K requests/sec, but not to 0. However, the throughput of SPaxos drops to 0 when leader fails. The latency of LaRSM only increases slightly, whereas the latency of SPaxos goes to infinity (Note that in the figure it is shown as around 500ms). This is because when leader fails, SPaxos stops ordering requests, thus no requests are handled by the system. For LaRSM, the clients which are connected to the failed replica, timeout on their current requests and then randomly connect to another replica. As discussed before, this timeout is usually much smaller than the timeout for suspecting a failure in SPaxos. Thus, the latency of a client in LaRSM only increases by a small amount. After the failure, the throughput of LaRSM remains around 16K requests/sec, which is because now there is one less replica in the system and the handling capability of the system decreases. For SPaxos, after a new leader is selected, the throughput increases to be a level slightly smaller than the throughput before the failure and the latency also decreases to be slightly higher than the latency before the failure. We also find that even though the throughput of LaRSM drops when a failure occurs, it still has better throughput than SPaxos, which indicates the better performance of LaRSM.

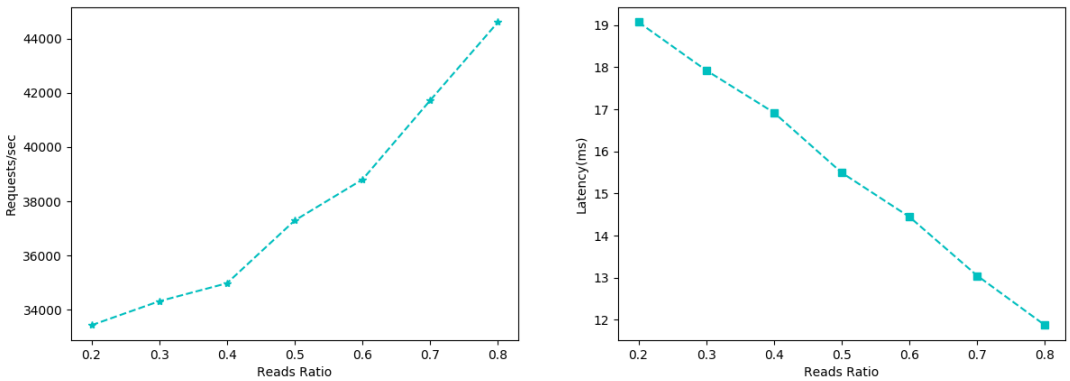


Figure 7 Throughput and Latency Under Different Reads Ratio.

6.3 Performance under Different Loads

In this part, we evaluate the performance of LaRSM on different types of work loads. This evaluation is done in a system of three replicas with 500 clients that keep issuing requests. We measure the throughput and latency as we increase the ratio of reads in a work load. The left part and right part of Fig. 7 give the throughput and latency change, respectively. It is shown in Fig. 7 that as the ratio of reads increases in a work load, the throughput of the system increases and the operation latency decreases. This confirms our optimization for the procedure to implement a linearizable RSM. As the reads ratio increases, the writes ratio decreases. Note that in a lattice agreement instance the input lattice is formed only by all the writes. When the number of writes is small, the proposal command set would be small and the message size would be small as well. Thus, the system can complete a lattice agreement instance faster. This shows that the performance LaRSM is even better for settings with fewer writes.

7 Conclusion

In this paper, we first give an algorithm to solve the lattice agreement problem in $O(\log f)$ rounds asynchronous rounds, which is an exponential improvement compared to previous $O(f)$ upper bound. We also give some optimizations for the GLA protocols proposed in literature. Evaluation results show that using lattice agreement to build a linearizable RSM has better performance than conventional consensus based RSM technique. Specifically, our implementation yields around 1.3x times throughput than SPaxos and incurs smaller latency in normal case.

References

1. Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM (JACM)*, 40(4):873–890, 1993.
2. Hagit Attiya, Maurice Herlihy, and Ophir Rachman. Atomic snapshots using lattice agreement. *Distributed Computing*, 8(3):121–132, 1995.
3. Martin Biely, Zarko Milosevic, Nuno Santos, and Andre Schiper. S-paxos: Offloading the leader for high throughput state machine replication. In *2012 IEEE 31st Symposium on Reliable Distributed Systems*, pages 111–120. IEEE, 2012.
4. Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407. ACM, 2007.
5. Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.
6. Jose M Faleiro, Sriram Rajamani, Kaushik Rajan, G Ramalingam, and Kapil Vaswani. Generalized lattice agreement. In *Proceedings of the 2012 ACM symposium on Principles of distributed computing*, pages 125–134. ACM, 2012.
7. Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. Technical report, Massachusetts Inst of Tech Cambridge lab for Computer Science, 1982.
8. Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.
9. Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.
10. Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.
11. Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.
12. Leslie Lamport and Mike Massa. Cheap paxos. In *International Conference on Dependable Systems and Networks, 2004*, pages 307–314. IEEE, 2004.
13. Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.
14. Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*, pages 386–400. Springer, 2011.
15. Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Convergent and commutative replicated data types. 2011.
16. Xiong Zheng, Vijay K Garg, and John Kaippallimalil. Linearizable replicated state machines with lattice agreement. *arXiv preprint arXiv:1810.05871*, 2018.
17. Xiong Zheng, Changyong Hu, and Vijay K Garg. Lattice agreement in message passing systems. In *32nd International Symposium on Distributed Computing (DISC 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

Exact Byzantine Consensus on Arbitrary Directed Graphs Under Local Broadcast Model

Muhammad Samir Khan
Department of Computer Science, University of Illinois at Urbana-Champaign, USA
mskhan6@illinois.edu

Lewis Tseng
Department of Computer Science, Boston College, USA
lewis.tseng@bc.edu

Nitin H. Vaidya
Department of Computer Science, Georgetown University, USA
nitin.vaidya@georgetown.edu

Abstract

We consider Byzantine consensus in a synchronous system where nodes are connected by a network modeled as a *directed graph*, i.e., communication links between neighboring nodes are not necessarily bi-directional. The directed graph model is motivated by wireless networks wherein asymmetric communication links can occur. In the classical point-to-point communication model, a message sent on a communication link is private between the two nodes on the link. This allows a Byzantine faulty node to *equivocate*, i.e., send inconsistent information to its neighbors. This paper considers the *local broadcast* model of communication, wherein transmission by a node is received identically by *all of its outgoing neighbors*, effectively depriving the faulty nodes of the ability to equivocate.

Prior work has obtained sufficient and necessary conditions on *undirected graphs* to be able to achieve Byzantine consensus under the *local broadcast* model. In this paper, we obtain tight conditions on *directed graphs* to be able to achieve Byzantine consensus with binary inputs under the *local broadcast* model. The results obtained in the paper provide insights into the trade-off between directionality of communication links and the ability to achieve consensus.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases complexity and impossibility results for distributed computing, fault-tolerance, reliability

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.30

Related Version A full version of the paper [14] is available on `arXiv:1911.07298`.

Funding This research is supported in part by the National Science Foundation awards 1409416 and 1733872, and Toyota InfoTechnology Center. 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.

1 Introduction

Byzantine consensus [22] is a classical problem in distributed computing. We consider a synchronous system consisting of n nodes, each with a *binary* input. The objective is for the nodes to reach consensus in the presence of up to f Byzantine faulty nodes. The nodes are connected by a communication network represented by a graph. In the classical point-to-point communication model, a message sent by a node to one of its neighboring nodes is received only by that node. This allows a Byzantine faulty node to send inconsistent messages to its neighbors without the inconsistency being observed by the neighbors. For instance, a faulty node u may report to neighbor v that its input is 0, whereas report to neighbor w that its input is 1. Node v will not hear the message sent by node u to node w. This ability of a faulty

© Muhammad Samir Khan, Lewis Tseng, and Nitin H. Vaidya;
licensed under Creative Commons License CC-BY
23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller; Article No. 30; pp. 30:1–30:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

node to send conflicting information on different communication links is called *equivocation* [4]. The problem of Byzantine consensus with equivocation in point-to-point networks is well-studied [2, 7, 17, 20, 22]. In undirected graphs, it is known that $n > 3f$ and connectivity $\geq 2f + 1$ are necessary and sufficient conditions to achieve Byzantine consensus [7].

This paper considers the *local broadcast* model of communication [3, 15]. In this model, a message sent by a node is received identically by all neighbors of that node. This allows the neighbors of a faulty node to detect its attempts to equivocate, effectively depriving the faulty node of the ability to send conflicting information to its neighbors. In the example above, in the local broadcast model, if node u attempts to send different input values to different neighbors, the neighbors will receive all the messages, and can detect the inconsistency. Recent work has shown that this ability to detect equivocation reduces network requirements for Byzantine consensus in *undirected* graphs. In particular, for the local broadcast model, recent work [13, 21] has identified that the following two conditions are both necessary and sufficient for Byzantine consensus in *undirected* graphs: network connectivity $\geq \lfloor 3f/2 \rfloor + 1$ and minimum node degree $\geq 2f$.

In this paper, we study Byzantine consensus in *directed* graphs under the local broadcast model. The directed graph model is motivated by wireless networks wherein asymmetric links may occur. Thus, the communication links between neighboring nodes are not necessarily bi-directional. Under local broadcast, when some node u transmits a message, it is received identically by all of u's *outgoing* neighbors (i.e., by nodes to whom there are outgoing links from node u). The results obtained in the paper provide insights into the trade-off between directionality of communication links and the ability to achieve consensus.

This paper makes two main contributions:

1. **Necessity:** In Section 5, we identify a necessary condition for directed graphs to solve Byzantine consensus. The proof is via a state machine based approach [2, 7, 8]. However, the communication by faulty nodes must follow the local broadcast model which restricts their behavior. The directed nature of the graph also adds to the difficulty. We handle this complexity via non-trivial arguments to show the desired result.
2. **Sufficiency:** In Section 6, we constructively show that the necessary condition is also sufficient by presenting a Byzantine consensus algorithm along with its proof of correctness. The key challenge for directed graphs is that communication may only exist in one direction between some pairs of nodes. Hence, it is not straightforward to adapt the prior algorithm [13] on undirected graphs. More specifically, in undirected graphs, each node has some path(s) to each of the other nodes.[1] Prior algorithm by Khan et. al. [13] utilizes this property to solve Byzantine consensus in undirected graphs under local broadcast. In the directed case, this property is not guaranteed due to the directionality of communication. We prove some non-trivial properties (Section 6.2) implied by the tight condition identified in this paper. This allows us to achieve consensus in a unique "source component" and then propagate that decision to the rest of the graph.

The rest of the paper is organized as follows. We discuss related work in Section 2. Section 3 formalizes the setting and introduces notation. Our main result is presented in Section 4. Necessity of the identified tight condition is shown in Section 5 while sufficiency is shown in Section 6. We summarize in Section 7.

[1] In undirected graphs, consensus is not possible if graph is not connected.

2 Related Work

Byzantine consensus is a well-studied problem [2, 7, 17, 20, 22] with tight conditions known for complete graphs [22], undirected graphs [7], and directed graphs [26] under the point-to-point communication model. For brevity, we focus here on related work that restricts equivocation by faulty nodes.

Rabin and Ben-Or [23] considered complete graphs with global broadcast under synchronous communication, while Clement et. al. [5] looked at non-equivocation in complete graphs under asynchronous communication. Amitanand et. al. [1] restricted equivocation by faulty nodes by partitioning, for each faulty node w, the remaining graph such that a message sent by w to any node is received identically by all nodes in the corresponding partition. However, the underlying graph in [1] is complete while we consider arbitrary directed graphs. Several works [9, 12, 24] have used *undirected* hypergraphs to model partial broadcast for the Byzantine consensus problem. In this model, a message sent on an hyperedge is received identically by all nodes in the hyperedge. The closest work to this paper is by Khan et. al. [13, 21], who obtained that minimum node degree $\geq 2f$ and network connectivity $\geq \lfloor 3f/2 \rfloor + 1$ are tight conditions for Byzantine consensus in *undirected* graphs under the local broadcast model. Here, we consider arbitrary *directed* graphs under the local broadcast model.

Restricted equivocation has also been used to study related problems. [6, 11, 10, 29] looked at reliability and privacy on partial broadcast networks. [3, 15, 16] have investigated the Byzantine *broadcast* problem under local broadcast on both undirected and directed graphs. In Byzantine broadcast, the goal is for a single source to transmit a binary value reliably throughout the network. We consider the Byzantine *consensus* problem, where the goal is for all nodes to agree on a common value.

Another line of work investigates iterative algorithms for *approximate* Byzantine consensus. In this problem, each node starts with a real value (or a vector of real values) and maintains a state variable. The updates are "memory less", i.e., the update rules allow a state update in each round to depend only on the current state and the state values received from neighbors. This problem has been investigated under the classical point-to-point communication model on directed graphs by Tseng and Vaidya [25] and Vaidya et. al. [27, 28], under partial broadcast modeled via directed hypergraphs by Li et. al. [19], and under the local broadcast model on directed graphs by LeBlanc et. al. [18] as well as by Zhang and Sundaram [30]. The network conditions are different than the ones presented in this paper, since the algorithm structure is restricted (as summarized above) in these prior works.

3 System Model and Notation

We consider a synchronous system consisting of n nodes. The communication network connecting the nodes is represented by a *directed* graph $G = (V, E)$, where $|V| = n$. Each of the n nodes is represented by a vertex $u \in V$. We interchangeably use the terms *node* and *vertex*. Every node in the graph knows the communication graph G. Each directed edge $(u, v) \in E$ represents a FIFO link from u to v. When a message m sent by node u on edge (u, v) is received by node v, node v knows that the message m was sent by node u. This assumption is implicit in the previous related work as well. We assume the *local broadcast* model of communication wherein a message sent by any node u is received identically and correctly by each node v such that $(u, v) \in E$.

A *Byzantine* faulty node may exhibit arbitrary behavior; however, its communication is still governed by the local broadcast model. We consider the *Byzantine consensus problem*.

Each node starts with a binary input and must output a binary value. There are at most $f > 0$ Byzantine faulty nodes in the system. The output at each node must satisfy the following conditions.

1. **Agreement:** All non-faulty nodes must output the same value.
2. **Validity:** The output of each non-faulty node must be an input of some non-faulty node.
3. **Termination:** All non-faulty nodes must decide on their output in finite time.

Neighborhood: If $(u, v) \in E$, then u is an *in-neighbor* of v and v is an *out-neighbor* of u. The *in-neighborhood* of a node v is the set of all in-neighbors of v, i.e., $\{u \mid (u, v) \in E\}$. Similarly, the *out-neighborhood* of a node v is the set of all out-neighbors of v, i.e., $\{u \mid (v, u) \in E\}$. In graph G, for node sets A and B, we define in-neighborhood of set B in set A, denoted $\Gamma_G(A, B)$, as the set of in-neighbors of nodes in B that are in set A. That is, $\Gamma_G(A, B) = \{u \in A \mid \exists v \in B \text{ s.t. } (u, v) \in E(G)\}$. Note that $E(G)$ denotes the set of edges in graph G. We will use the above definition for different graphs, hence the subscript G above is important. We may drop the subscript G when it is clear from the context.

We will say that $A \rightarrow_G B$ if $|\Gamma_G(A, B)| > f$. Here as well, we may drop the subscript G when it is clear from the context.

Paths in graph G: A path is a sequence of distinct nodes such that if u precedes v in the sequence, then u is an in-neighbor of v in G (i.e., (u, v) is an edge).

- For two nodes u and v, a uv-path P_{uv} is a path from u to v. u is called the *source* and v the *terminal* of P_{uv}. Any other node in the path is called an *internal* node of P_{uv}. Two uv-paths are *node-disjoint* if they do not share a common internal node.
- For a set $U \subsetneq V$ and a node $v \notin U$, a Uv-path is a uv-path for some node $u \in U$. All Uv-paths have v as the terminal. Two Uv-paths are *node-disjoint* if they do not have any nodes in common except terminal node v. In particular, two node-disjoint Uv-paths have different source nodes.

A path is said to <u>*exclude*</u> a set of nodes $X \subset V$ if no internal node of the path belongs to X; however, its source and terminal nodes may potentially belong to X. A path is said to be <u>*fault-free*</u> if none of its internal nodes are faulty. In other words, a path is fault-free if it excludes the set of faulty nodes. Note that a fault-free path may have a faulty node as either source or terminal.

We use the notation $A \overset{X}{\rightsquigarrow}_G B$ if, for every node $u \in B$, there exist at least $f + 1$ node-disjoint Au-paths in G that exclude X, i.e., there exist $f + 1$ node-disjoint Au-paths that have only u in common and none of them contain any internal node from the set X. We may omit the subscript G when it is clear from the context.

With a slight abuse of terminology, we allow a *partition* of a set to have empty parts. That is, $(Z_1, \ldots, Z_k)$ is a partition of a set Y if $\cup_{i=1}^{k} Z_i = Y$ and $Z_i \cap Z_j = \emptyset$ for all $i \neq j$, but some Z_i's can be possibly empty.

For a set of nodes $U \subsetneq V$,

- $G[U]$ is the subgraph induced by the nodes in U.
- G_{-U} is the graph obtained from G by removing all edges (v, u) such that $u \in U$, i.e., by removing all incoming edges to U. Observe that if P is a path in G_{-U}, then P is a path in G that excludes U and terminates in $V - U$. Conversely, if P is a path in G that excludes U and terminates in $V - U$, then P is a path in G_{-U}.

A directed graph G is *strongly connected* if for each pair of nodes u, v, there is both a uv-path and a vu-path in G. A *directed graph decomposition* of G is a partition of G into

non-empty parts $H_1, \ldots, H_k$, where $k > 0$, and each H_i is a maximal strongly connected subgraph of G – each H_i is assumed to be maximal in the sense that adding any nodes to H_i will destroy its strong connectivity. Let $\mathcal{H}$ be the graph obtained from the decomposition by contracting each H_i into a node c_i, so that there is an edge (c_i, c_j) in $\mathcal{H}$ if there is an edge from a node in H_i to a node in H_j in G. Then, graph $\mathcal{H}$ is acyclic. If a node c_i has no in-neighbors, then H_i is called a *source component* of the decomposition. Note that, since $\mathcal{H}$ is acyclic, there is always at least one source component of a directed graph decomposition.

4 Main Results

The main result of this paper is a tight network condition for consensus in directed graphs under the local broadcast model. The following definition presents the condition and the accompanying theorem states the result.

▶ **Definition 1.** *A directed graph G satisfies* condition SC with parameter F *if for every partition (A, B) of V, where both $A - F$ and $B - F$ are non-empty, we have that either $A \overset{F}{\rightsquigarrow} B - F$ or $B \overset{F}{\rightsquigarrow} A - F$. We say that G satisfies* condition SC*, if G satisfies condition SC with parameter F for every set $F \subseteq V$ of cardinality at most f.*

▶ **Theorem 2.** *Under the local broadcast model, Byzantine consensus tolerating at most f Byzantine faulty nodes is achievable on a directed graph G if and only if G satisfies condition SC.*

Proof. The proof follows from Theorems 4, 5, and 11 presented later. ◀

Intuitively, the above condition requires that at least one of the two partitions A and B should have the ability to "propagate" its state to the other partition reliably. For the point-to-point communication model, Tseng and Vaidya [26] obtained an analogous network condition, which is that, for every partition (A, B) of V and a faulty set F, where both $A - F$ and $B - F$ are non-empty, either $A - F \overset{F}{\rightsquigarrow} B - F$ or $B - F \overset{F}{\rightsquigarrow} A - F$. In the point-to-point communication model a faulty node can equivocate. Thus, the condition in [26] does not allow nodes in set F to be source nodes,[2] and requires $A - F$ or $B - F$ to propagate its state to the non-faulty nodes in the other partition. On the other hand, as discussed earlier, local broadcast effectively removes a faulty node's ability to equivocate. Therefore, the condition in Definition 1 allows a node in set F to be a source node in the propagation paths, but does not allow nodes in F to be internal nodes on such paths.

Even though the conditions for point-to-point communication [26] and local broadcast seem similar, the algorithm in [26] is not immediately adaptable to the local broadcast model. One key challenge is that while non-equivocation provided by local broadcast prevents a faulty node from sending conflicting messages in one round, it does not directly stop a faulty node from lying inconsistently across rounds, even to the same neighbor. We discuss this at the end of Section 6.1.

We prove necessity of condition SC via a state machine based approach [2, 7, 8] similar to the proofs of necessity in [13, 26]. However, care must be taken to ensure that we do not break the local broadcast property. The formal proof is given in the full version of the paper [14] – since it is somewhat difficult. In Section 5, we provide an intuitive sketch of the proof. The sufficiency is proved constructively. In Section 6, we present an algorithm to achieve consensus when the communication graph satisfies condition SC, accompanied by a proof of correctness.

[2] Recall that, in a uv-path, u is the source node and v is the terminal.

5 Necessity

In this section, we show that condition SC (Definition 1) is necessary for consensus. We first present another property, condition NC. This condition is equivalent to condition SC, as stated in the next theorem, and we will use it to prove necessity in Theorem 5. Recall that we use $A \rightarrow_G B$ to denote $|\Gamma_G(A, B)| > f$ (subscript G is dropped when clear from context).

▶ **Definition 3.** *A directed graph G satisfies* condition NC with parameter F *if for every partition (L, C, R) of V, where both $L - F$ and $R - F$ are non-empty, we have that either $R \cup C \rightarrow_G L - F$ or $L \cup C \rightarrow_G R - F$. We say that G satisfies* condition NC*, if G satisfies condition NC with parameter F for every set $F \subseteq V$ of cardinality at most f.*

▶ **Theorem 4.** *A directed graph G satisfies condition NC if and only if G satisfies condition SC.*

A formal proof of Theorem 4 appears in the full paper [14]. The following theorem states that condition NC is necessary for consensus. Since condition NC and condition SC are equivalent, as a corollary we get the necessity part of Theorem 2.

▶ **Theorem 5.** *If there exists a Byzantine consensus algorithm under the local broadcast model on a directed graph G tolerating at most f Byzantine faulty nodes, then G satisfies condition NC.*

As mentioned earlier, the formal proof of this theorem is given in the full paper [14]. Here we give a sketch of the proof. Suppose for the sake of contradiction that there exists an algorithm that solves Byzantine consensus under the local broadcast model on a graph G which does not satisfy condition NC. Then there exists a set F of cardinality at most f and a partition (L, C, R) of G, where both $L - F$ and $R - F$ are non-empty, such that $R \cup C \not\rightarrow L - F$ and $L \cup C \not\rightarrow R - F$. We create three executions E_1, E_2, and E_3 using the algorithm as follows.

- $\boldsymbol{E_1}$: $\Gamma(R \cup C, L - F)$, the in-neighborhood of $L - F$ in $R \cup C$, is the faulty set. We partition the faulty set into two parts: the in-neighborhood of $L - F$ in $R - F$ and C, $\Gamma((R-F) \cup C, L-F)$, and the in-neighborhood of $L-F$ in $R \cap F$, $\Gamma(R \cap F, L-F)$. Both these sets have different behavior. In each round, a faulty node in $\Gamma((R-F) \cup C, L-F)$ broadcasts the same messages as the corresponding non-faulty node in E_3, while a faulty node in $\Gamma(R \cap F, L - F)$ broadcasts the same messages as the corresponding non-faulty node in E_2. All non-faulty nodes have input 0. So by validity, all non-faulty nodes decide on output 0 in finite time.
- $\boldsymbol{E_2}$: $\Gamma(L \cup C, R - F)$, the in-neighborhood of $R - F$ in $L \cup C$, is the faulty set. We partition the faulty set into two parts: the in-neighborhood of $R - F$ in $L - F$ and C, $\Gamma((L-F) \cup C, R-F)$, and the in-neighborhood of $R-F$ in $L \cap F$, $\Gamma(L \cap F, R-F)$. Both these sets have different behavior. In each round, a faulty node in $\Gamma((L-F) \cup C, R-F)$ broadcasts the same messages as the corresponding non-faulty node in E_3, while a faulty node in $\Gamma(L \cap F, R - F)$ broadcasts the same messages as the corresponding non-faulty node in E_1. All non-faulty nodes have input 1. So by validity, all non-faulty nodes decide on output 1 in finite time.
- $\boldsymbol{E_3}$: $F \cap (L \cup R)$ is the faulty set. We partition the faulty set into two parts: $F \cap L$ and $F \cap R$. Both these sets have different behavior. In each round, a faulty node in $F \cap L$ broadcasts the same messages as the corresponding non-faulty node in E_1, while a faulty node in $F \cap R$ broadcasts the same messages as the corresponding non-faulty node in

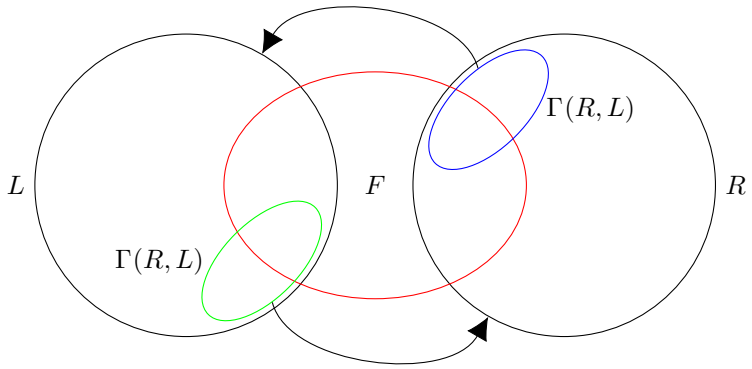


Figure 1 Necessity proof: the simple case when $C = \emptyset$. The nodes in blue, green, and red depict the three faulty sets for executions E_1, E_2, and E_3 respectively.

E_2. All non-faulty nodes in L have input 0 and all non-faulty nodes in $R \cup C$ have input 1. The output of the non-faulty nodes in this execution will be described later.

In the full paper [14] we make the above description of the three executions precise. For the simple case when $C = \emptyset$, Figure 1 depicts the faulty nodes in the three executions.

To see the output of non-faulty nodes in E_3, note that non-faulty nodes in $L - F$ receive the same messages in each round, from their in-neighbors, as the corresponding nodes in E_1. They also have the same input, 0, so they decide on the same output in both the executions, i.e., 0. Similarly, the non-faulty nodes in $R - F$ receive the same messages in each round, from their in-neighbors, as the corresponding nodes in E_2. They also have the same input, 1, so they decide on the same output in both the executions, i.e., 1. Since both $L - F$ and $R - F$ are non-empty, this violates agreement, a contradiction.

6 Sufficiency

In this section, we constructively prove the sufficiency portion of Theorem 2. Together with the necessity result shown in Theorem 5, we have that this result is tight. We present a Byzantine consensus algorithm in Section 6.1. The algorithm utilizes some non-trivial graph properties implied by condition SC. We show these in Section 6.2. In Section 6.3, we give a proof of correctness of the algorithm, assuming that the graph G satisfies condition SC.

6.1 Algorithm

Algorithm 1 presents pseudocode for the proposed algorithm. Each node $v \in V$ maintains a local state variable named γ_v. At the beginning of the algorithm, this is initialized to equal node v's binary input. γ_v is modified during the execution of the algorithm. The output of each node v is the value of its state variable γ_v at the end of the algorithm. The algorithm execution is viewed as being divided into *phases*, each phase consisting of one iteration of the `for` loop in the pseudocode. Each phase has an associated distinct subset $F \subseteq V$ of cardinality at most f.

The algorithm draws inspiration from the strategy used in the algorithms in [13] and [26]. However, the details are significantly different, which we discuss at the end of this section. In particular, each phase (i.e., each iteration of the `for` loop) in the algorithm considers a candidate faulty set F, and the nodes attempt to reach consensus by the end of that phase

assuming that F is indeed the set of all the faulty nodes. Let F^* denote the set of nodes that are actually faulty in a given execution of the algorithm. Then each node updates its state variable in such a manner so that

(i) when $F = F^*$, the state variable at all non-faulty nodes is identical at the end of this phase, i.e., all non-faulty nodes reach consensus in this phase (Lemma 12), and

(ii) when $F \neq F^*$, the value of the state variable at a non-faulty node at the end of the phase equals the state of some non-faulty node at the start of the phase (Lemma 13).

The first objective ensures that the nodes reach agreement. The second inductively implies that this agreement, once achieved, is not lost, as well as that the nodes decide on an input of some non-faulty node, i.e., validity. Termination follows from the fact that there are only a finite number of executions.

We now discuss the steps performed by each node v in a given phase. Some of the steps of the algorithm are based on those in [13] and are explained here again for completeness.

Graph decomposition: In step (a) of a given phase, each node performs a directed graph decomposition on $G - F$. Recall that we assume that each node knows the topology of graph G, so each node can perform this step locally. Since G satisfies condition SC with parameter F, it turns out that $G - F$ has a unique source component S (Lemma 6). The rest of the steps in the phase are aimed at nodes in S attempting to agree on some common value, and then propagating that value to the rest of the graph.

Flooding: In Step (b), nodes in S and their in-neighbors in F, $\Gamma(F, S)$, flood the value of their γ state variables. The "flooding" procedure used here is analogous to that in [13] for undirected graphs. Without much modification, it can be adapted for directed graphs. This procedure is presented in the full version of the paper [14].

Consider a node $u \in S \cup \Gamma(F, S)$. In the flooding procedure, u attempts to transmit its state variable to every node v such that there exists a uv-path in G. At the end of the flooding procedure, for each such node v and a uv-path P_{uv}, v will have received a binary value b along P_{uv}. If P_{uv} is fault-free, then $b = \gamma_u$, the state variable of u. However, if P_{uv} is not fault-free, then an intermediary faulty node may tamper with the messages, so it is possible that $b \neq \gamma_u$.

Consensus in the source component S: Next, using steps (c) and (d), the nodes in the unique source component try to reach consensus. Based on the values flooded by each node, set $S \cup \Gamma(F, S)$ can be partitioned into nodes that flooded 0, namely set Z, and nodes that flooded 1, namely set N. In step (c), node v attempts to estimate the sets Z and N using the values received on paths excluding F, i.e., ignoring paths that have nodes from F as intermediaries. When $F \neq F^*$, faulty nodes may tamper the messages and v may incorrectly categorize some nodes. However, when $F = F^*$, all non-faulty nodes correctly determine Z and N.

In particular, in step (c), each node $v \in S$ partitions $S \cup \Gamma(F, S)$ as follows. Recall that a path is said to exclude set F if none of its *internal* nodes are in F. For each $u \in S \cup \Gamma(F, S)$, node v chooses an arbitrary uv-path P_{uv} that excludes F. It can be shown (Lemma 8) that such a path always exists. For the purpose of step (c), node v is deemed to have received its own γ_v value along path P_{vv} (containing only node v). In step (c), as shown in the pseudo-code, node v partitions $S \cup \Gamma(F, S)$ into sets Z_v and N_v, its estimates of sets Z and N, based on values received along the above paths.

Step (d) specifies the rules for updating γ_v value. γ_v is not necessarily updated in each phase. If $F = F^*$, then all nodes in S will have the same γ_v value after this step (Lemma 12). That is, in the phase in which $F = F^*$, the nodes in S achieve consensus in step (d).

Algorithm 1 Proposed algorithm for Byzantine consensus under the local broadcast model in directed graphs: Steps performed by node v are shown here.

Each node v has a binary input value in $\{0,1\}$ and maintains a binary state $\gamma_v \in \{0,1\}$.
Initialization: $\gamma_v :=$ input value of node v.
For *each* $F \subseteq V$ *such that* $|F| \leq f$ **do**

Step (a): Perform directed graph decomposition on $G - F$. Let S be the unique source component (Lemma 6).

Step (b): If $v \in S \cup \Gamma(F,S)$, then flood value γ_v (the steps taken to achieve flooding are described in the full paper [14]).

Step (c): If $v \in S$, for each node $u \in S \cup \Gamma(F,S)$, identify a single uv-path P_{uv} that excludes F. Let,

$$Z_v := \left\{u \in S \cup \Gamma(F,S) \mid v \text{ received value 0 from } u \text{ along } P_{uv} \text{ in step (b)}\right\},$$
$$N_v := S \cup \Gamma(F,S) - Z_v.$$

Step (d): If both $Z_v - F$ and $N_v - F$ are non-empty, then

If $Z_v \overset{F}{\rightsquigarrow} N_v - F$,
then set $A_v := Z_v$ and $B_v := N_v - F$,
else set $A_v := N_v$ and $B_v := Z_v - F$.

If $v \in B_v$ and v received value $\delta \in \{0,1\}$, in step (b), identically along any $f+1$ node-disjoint $A_v v$-paths that exclude F, then set $\gamma_v := \delta$.

Step (e): If $v \in S$, then flood value γ_v.

Step (f): If $v \in V - S - F$ and v received value $\delta \in \{0,1\}$, in step (e), identically along any $f+1$ node-disjoint Sv-paths that exclude F, then set $\gamma_v := \delta$.

end
Output γ_v.

Propagating decision to rest of the graph: Since we iterate over all possible faulty sets, in one phase of the algorithm, F is correctly chosen to be exactly F^*, the set of actual faulty nodes. In this phase, nodes in S will reach consensus by step (d). In steps (e) and (f), nodes in S propagate their state to the rest of the graph. In particular, in step (e), nodes in S flood their γ_v value, as in step (b), while step (f) specifies the rules for nodes in $V - S - F$ to update their γ_v state variable.

Output: After all the phases (i.e., all iterations of the `for` loop) are completed, the value of γ_v is chosen as the output of node v.

As mentioned earlier, the proposed algorithm uses the same strategy as in [13] and [26] by iterating over all possible faulty sets. However, the steps performed in each iteration are significantly different than both.

- The algorithm for directed graphs under point-to-point communication in [26] is not immediately adaptable to the local broadcast setting. One key challenge is that it requires nodes to send messages in multiple rounds in a single iteration of the main `for` loop. While non-equivocation provided by local broadcast prevents a faulty node from sending conflicting messages in one round, it does not directly stop a faulty node from lying inconsistently across rounds, even to the same neighbor. In our algorithm, when $F = F^*$, the faulty nodes are only allowed to flood their states once in the entire iteration, preventing them from lying inconsistently across rounds in this iteration.

- On the other hand, the algorithm for undirected graphs under local broadcast [13] does indeed require each faulty node to send a single message for each iteration of the main for loop. However, in directed graphs, in contrast with the undirected setting, communication may only exist in one direction between some pairs of nodes. Our algorithm first achieves consensus in a unique source component and then propagates this to the rest of the graph. The existence of a *unique* source component or its ability to propagate its state to the rest of the graph is not immediately obvious. In the next section, we show that these and other non-trivial properties are guaranteed by condition SC, which our algorithm utilizes to solve consensus.

6.2 Graph Properties

Algorithm 1 relies on some non-trivial properties of graphs that satisfy condition SC. In this section, we prove these properties followed by the proof of correctness of the algorithm in Section 6.3. We first show that there is indeed a unique source component in the directed graph decomposition of $G - F$, performed in step (a) of each phase. Suppose, for the sake of contradiction, that there are two source components S_1 and S_2 of the decomposition. Note that, by construction, there are no edges into S_1 and S_2, except from F. By appropriately selecting sets A and B, the only paths into A (resp. B) are via F and so are limited to at most f, violating condition SC with parameter F, a contradiction.

▶ **Lemma 6.** *For any choice of set F in the algorithm the directed graph decomposition of $G - F$ has a unique source component.*

Proof. Fix an arbitrary set F. Suppose for the sake of contradiction that $G - F$ has two source components S_1 and S_2. Let $C := V - S_1 - S_2$ be the rest of the nodes. We create a partition (A, B) that violates the requirements of condition SC with parameter F (Definition 1). Let $A := S_1$ and $B := S_2 \cup C = V - S_1$. First note that both $A - F = A = S_1$ and $B - F \supseteq S_2$ are non-empty.

Now, by construction, we have that S_2 has no incoming edges except from F. Therefore, F is a cut set that separates S_2 from S_1, i.e., there are no paths from a node in S_1 to a node in S_2 in $G - F$. By Menger's Theorem, there are at most $|F| \leq f$ node-disjoint paths in G from $S_1 = A$ to any node in $S_2 \subseteq B - F$. Therefore, for any node $v \in S_2 \subseteq B - F$, there are at most f node-disjoint Av-paths that exclude F, and so $A \overset{F}{\not\rightarrow}_G B - F$. Similarly, by construction, we have that S_1 has no incoming edges except from F. Therefore, F is a cut set that separates S_1 from $S_2 \cup C - F$. By Menger's Theorem, there are at most $|F| \leq f$ node-disjoint paths in G from $S_2 \cup C = B$ to any node in $S_1 = A - F$. Therefore, for any node $v \in S_1 = A - F$, there are at most f node-disjoint Bv-paths that exclude F, and so $B \overset{F}{\not\rightarrow}_G A - F$. This violates condition SC, a contradiction. ◀

Next, this unique source component S, along with its in-neighborhood in F, satisfies condition SC with parameter F. We follow the same approach as in proof of Lemma 6, assuming for the sake of contradiction that $G[S \cup \Gamma(F, S)]$ does not satisfy condition SC with parameter F, and showing that this implies that G does not satisfy condition SC with parameter F.

▶ **Lemma 7.** *For any choice of set F in the algorithm, let S be the unique source component of $G - F$. Then $G[S \cup \Gamma(F, S)]$ satisfies condition SC with parameter F.*

Proof. Suppose for the sake of contradiction that $H := G[S \cup \Gamma(F, S)]$ does not satisfy condition SC with parameter F. So there exists a partition (A, B) of $S \cup \Gamma(F, S)$, such that

$A - F$ and $B - F$ are non-empty, and $A \overset{F}{\not\rightarrow}_H B - F$ and $B \overset{F}{\not\rightarrow}_H A - F$. Let the rest of the nodes in G be denoted by $C := V - S \cup \Gamma(F, S)$. Let $A' := A \cup C$. Then (A', B) is a partition of V, with $A' - F$ and $B - F$ both non-empty.

We first show that there is no edge from C to S in G. Observe that C is disjoint from $S \cup \Gamma(F, S)$. If a node $u \in C \cap F$ has an edge to a node in S, then $u \in \Gamma(F, S)$, a contradiction since $C \cap \Gamma(F, S) = \emptyset$. On the other hand, if $u \in C - F$ has an edge to a node in S, then the edge exists in $G - F$, which is a contradiction since S is a source component in $G - F$.

Now, since $A \overset{F}{\not\rightarrow}_H B - F$, we have that for some node $v \in B - F$ there are at most f node-disjoint Av-paths in H_{-F}.[3] By Menger's Theorem, there exists a cut set $X \not\ni v$ of cardinality at most f that separates v from $A - X$ in H_{-F}. Since there is no edge from C to S in G, we have that X also separates v from $(A - X) \cup C = A' - X$ in G_{-F}. It follows that there are at most f node-disjoint $A'v$-paths that exclude F in G, and so $A' \overset{F}{\not\rightarrow}_G B - F$.

Similarly, since $B \overset{F}{\not\rightarrow}_H A - F$, we have that for some node $v \in A - F$ there are at most f node-disjoint Bv-paths in H_{-F}. By Menger's Theorem, there exists a cut set $X \not\ni v$ of cardinality at most f that separates v from $B - X$ in H_{-F}. Since there is no edge from C to S in G, we have that X also separates v from $B - X$ in G_{-F}. Note that $v \in A - F \subseteq A' - F$. It follows that there are at most f node-disjoint Bv-paths that exclude F in G, and so $B \overset{F}{\not\rightarrow}_G A' - F$. This violates condition SC, a contradiction. ◀

We now show that the paths identified in steps (c), (d), and (e) of the algorithm do indeed exist. The existence of paths in step (c) follows by construction of S.

▶ **Lemma 8.** *For any choice of set F in the algorithm, let S be the unique source component of $G - F$. Then, for any two nodes $u \in S \cup \Gamma(F, S)$ and $v \in S$, there exists a uv-path that excludes F.*

Proof. If $u \in S$, then there exists a uv-path in $G - F$ since S is strongly connected in $G - F$ by construction. If $u \in \Gamma(F, S)$, then there exists a node $w \in S$ such that (u, w) is an edge in G. Also, there exists a wv-path P_{wv} in $G - F$ since S is strongly connected in $G - F$. So u - P_{wv} is a uv-path in G that excludes F. ◀

The existence of paths in step (d) follows from Lemma 7 and definition of condition SC (Definition 1).

▶ **Lemma 9.** *For any non-faulty node v, and any given phase with the corresponding set F, in step (d), if $v \in B_v$, then there exist $f + 1$ node-disjoint $A_v v$-paths that exclude F.*

Proof. Fix a phase of the algorithm and the corresponding set F. Consider an arbitrary non-faulty node v such that $v \in B_v$ in step (d). By construction, B_v is either $N_v - F$ or $Z_v - F$, both of which are non-empty. In the first case, we have that $A_v = Z_v \overset{F}{\rightsquigarrow} N_v - F = B_v$. In the second case, we have that $Z_v \overset{F}{\not\rightarrow} N_v - F$. By Lemma 7, we have that $G[Z_v \cup N_v] = G[S \cup \Gamma(F, S)]$ satisfies condition SC with parameter F. Therefore $A_v = N_v \overset{F}{\rightsquigarrow} Z_v - F = B_v$. ◀

[3] Recall from Section 3 that G_{-U} is the graph obtained from G by removing all incoming edges to U so that if P is a path in G_{-U}, then P is a path in G that excludes U and terminates in $V - U$, and if P is a path in G that excludes U and terminates in $V - U$, then P is a path in G_{-U}.

For paths in step (e), note that, by construction of S, the in-neighbors of S are contained entirely in F. So there can only be at most f paths into S from $V - S$. Since G satisfies condition SC with parameter F, we get that $S \overset{F}{\rightsquigarrow} V - S - F$.

▶ **Lemma 10.** *For any choice of set F in the algorithm, let S be the unique source component of $G - F$. Then $S \overset{F}{\rightsquigarrow} V - S - F$.*

Proof. Let $A = S$ and $B = F \cup (V - S) = V - A$. Now, since $A = S$ is the unique source component of $G - F$, we have that $\Gamma(B, A) \subseteq F$, i.e., nodes in F are the only ones in B to have an edge into A. So B can have at most f node-disjoint paths to any node in A. Thus $B \overset{F}{\not\rightsquigarrow} A - F$. Since G satisfies condition SC, we have that $S = A \overset{F}{\rightsquigarrow} B - F = V - S - F$, as required. ◀

6.3 Proof of Correctness

We provide a proof of correctness of Algorithm 1 by proving Theorem 11 below.

▶ **Theorem 11.** *Under the local broadcast model, Byzantine consensus tolerating at most f Byzantine faulty nodes is achievable on a directed graph G, if G satisfies condition SC.*

Let us assume that G satisfies condition SC. For convenience, we will refer to the state variable γ_v as the "state of node v". We remind the reader that the algorithm proceeds in phases and each phase has an associated unique set F, with $|F| \leq f$. We use F^* to denote the actual set of faulty nodes in a given execution.

As mentioned earlier, the algorithm attempts to balance two objectives. We formalize them in the two lemmas below. The proofs of Lemmas 12 and 13 are presented later.

▶ **Lemma 12.** *Consider the unique phase of Algorithm 1 where the corresponding set $F = F^*$. At the end of this phase, every pair of non-faulty nodes $u, v \in V$ has an identical state, i.e., $\gamma_u = \gamma_v$.*

▶ **Lemma 13.** *For a non-faulty node v, its state γ_v at the end of any given phase equals the state of some non-faulty node at the start of that phase.*

The correctness of Algorithm 1 follows from these two lemmas, as shown next.

Proof of Theorem 11. To prove the correctness of Algorithm 1, we have to prove the three properties of Agreement, Validity, and Termination, as specified in Section 3.

Termination: The algorithm satisfies the termination property because there are a finite number of phases in the algorithm, and each of them completes in finite time.

Agreement: The total number of faulty nodes is bounded by f. Therefore, in any execution, there exists at least one phase in which the set $F = F^*$. From Lemma 12, all non-faulty nodes have the same state at the end of this phase. Lemma 13 implies that the state of the non-faulty nodes will remain unchanged in the subsequent phases. So all non-faulty nodes will have identical outputs, satisfying the agreement property.

Validity: The state of each non-faulty node is initialized to its own input at the start of the algorithm. So the state of each non-faulty node is its own input at the start of the first phase. By applying Lemma 13 inductively, we have that the state of a non-faulty node always equals the *input* of some non-faulty node. This satisfies the validity property.

This completes the proof of correctness of Algorithm 1. ◀

The rest of the section focuses on proving Lemmas 12 and 13. We assume that the graph G satisfies condition SC, even if it is not explicitly stated. We use F^* to denote the actual faulty set. The following observation follows from the rules used for flooding [14].

▶ **Observation 14.** *For any phase of Algorithm 1, for any two nodes $u, v \in V$ (possibly faulty), if v receives value b along a fault-free uv-path then u broadcast the value b to its neighbors during flooding.*

Fix a phase in the algorithm along with the corresponding set F. For Lemma 12, the correctness relies on the local broadcast property. Suppose, as stated in the statement of the lemma, that $F = F^*$ in a given phase. There are two cases to consider for any non-faulty node v. In the first case $v \in S$. In this case, the paths used by v in step (c) of the phase exclude F. So these paths are fault-free (i.e., none of their internal nodes are faulty). Then, the properties of flooding imply that any two non-faulty nodes u, v will obtain $Z_u = Z_v$ and $N_u = N_v$ in step (c). By a similar argument, all the paths used in step (d) of this phase are also fault-free, and any two non-faulty nodes will end step (d) with an identical state. In the second case $v \in V - S - F$, a repeat of the above argument implies that the paths used in step (f) are both fault-free and have non-faulty source nodes. Since the source nodes are all from S, they have an identical state at the end of step (d). So v correctly updates its state in step (f) to match the identical state in S.

Proof of Lemma 12. Fix a phase of the algorithm and the corresponding set F such that $F = F^*$. We first show that all nodes in S have identical state, at the end of the phase, and then consider nodes in $V - S - F^*$.

Let Z be the set of nodes that flooded 0 in step (b) and let N be the set of nodes that flooded 1 in step (b). Note that Z and N may contain faulty nodes, but due to the broadcast property, a faulty node is in at most one of these sets. Consider any non-faulty node $v \in S$. Then $Z_v = Z$ and $N_v = N$, as follows. Let $w \in S \cup \Gamma(F^*, S)$ be an arbitrary node that flooded 0 (resp. 1) in step (b), i.e., $w \in Z$ (resp. $w \in N$). Now the wv-path P_{wv} identified by v in step (b) excludes F^* and is fault-free. So, by Observation 14, v receives 0 (resp. 1) along P_{wv} and correctly sets $w \in Z_v$ (resp. $w \in N_v$).

Therefore, we have that for any two non-faulty nodes $u, v \in S$, $Z_u = Z_v = Z$ and $N_u = N_v = N$. If $Z - F^*$ (resp. $N - F^*$) is empty, then $S = N - F^*$ (resp. $S = Z - F^*$). So all non-faulty nodes in S have identical state, which is not updated, and the claim is trivially true. So suppose both $Z - F^*$ and $N - F^*$ are non-empty. Since $Z_u = Z_v$ and $N_u = N_v$, we have that $A_u = A_v$ and $B_u = B_v$. Let $A := A_u$ and $B := B_u$. By construction $A \overset{F^*}{\rightsquigarrow} B$. Now all nodes in A flooded identical value in step (b), say α. If $u \in A$, then u's state is α at the beginning of the phase and stays unchanged in step (d), i.e., $\gamma_u = \alpha$. If $u \in B$, then the $f+1$ node-disjoint Au-paths identified by u in step (d) exclude F^* and so are all fault-free. By Observation 14, it follows that u receives α identically along these $f+1$ paths and updates $\gamma_u = \alpha$. Similarly, $\gamma_v = \alpha$.

So we have shown that all non-faulty nodes in S have identical state α at the end of step (d). Since these nodes do not update their state in the rest of the phase, so we have that all nodes in S have state α at the end of the phase. Since all nodes in S are non-faulty, we have that each node in S floods α in step (e). Consider any non-faulty node $v \in V - S - F^*$. The $f+1$ node-disjoint Sv-paths identified by v in step (f) exclude F^* and so are all fault-free. Recall that S contains only nodes in $V - F^*$, i.e., only non-faulty nodes, so the source nodes in these $f+1$ paths are also non-faulty. By Observation 14, it follows that v receives α identically along these $f+1$ paths and updates $\gamma_v = \alpha$. So by the end of the phase, all non-faulty nodes have identical state α, as required. ◀

Lemma 13 follows from the following observation. In steps (d) and (f), if a node v updates its state, then it must have received identical value along $f+1$ node-disjoint $A_v v$-paths. Therefore, at least one of these paths must both be fault-free and have a non-faulty source.

Proof of Lemma 13. Fix a phase of the algorithm and the corresponding set F. We use γ_u^{start} and γ_u^{end} to denote the state γ_u of node u at the beginning and end of the phase, respectively. Let v be an arbitrary non-faulty node. If v does not update its state in this phase, then $\gamma_u^{\text{end}} = \gamma_u^{\text{start}}$ and the claim is trivially true. So suppose that v did update its state in this phase. This implies that $v \in V - F$.

There are now two cases to consider

Case 1: $v \in S$. It follows that v updated its state in step (d). Therefore, $v \in B_v$ and v received identical values along $f+1$ node-disjoint $A_v v$-paths in step (b). Now, at least one of these paths 1) is fault-free, and 2) has a non-faulty source u. By Observation 14, we have that the value received by v along this uv-path in step (b) is the value flooded by u in step (b). So $\gamma_v^{\text{end}} = \gamma_u^{\text{start}}$, where u is a non-faulty node.

Case 2: $v \in V - S - F$. It follows that v updated its state in step (f). Therefore, v received identical values along $f+1$ node-disjoint Sv-paths in step (e). Now, at least one of these paths 1) is fault-free, and 2) has a non-faulty source $w \in S$. By Observation 14, we have that the value received by v along this wv-path in step (e) is the value flooded by w in step (e). Note that $w \in S$ and so the value flooded by w in step (e) is the state of w at the end of this phase, i.e., w flooded γ_w^{end} in step (e). There are further two cases to consider

Case i: w did not update its state in this iteration. Then $\gamma_v^{\text{end}} = \gamma_w^{\text{end}} = \gamma_w^{\text{start}}$. Recall that w is a non-faulty node.

Case ii: w did update its state in this iteration. Then, from Case 1 above, we have that there exists a non-faulty node u such that $\gamma_w^{\text{end}} = \gamma_u^{\text{start}}$. So $\gamma_v^{\text{end}} = \gamma_w^{\text{end}} = \gamma_u^{\text{start}}$, where u is a non-faulty node.

In all cases, we have that γ_v^{end} equals γ_u^{start} for some non-faulty node u. ◀

7 Summary

In this work, we have presented tight conditions for exact binary Byzantine consensus in directed graphs under the local broadcast model. The sufficiency proof in Section 6 is constructive. However, the algorithm has exponential round complexity. We leave finding a more efficient algorithm for future work. The following question is also open: does there exist an efficient algorithm to check if a given directed graph satisfies condition SC?

References

1. S. Amitanand, I. Sanketh, K. Srinathant, V. Vinod, and C. Pandu Rangan. Distributed Consensus in the Presence of Sectional Faults. In *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*, PODC '03, pages 202–210, New York, NY, USA, 2003. ACM. doi:10.1145/872035.872065.
2. Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics.* John Wiley & Sons, Inc., USA, 2004.
3. Vartika Bhandari and Nitin H. Vaidya. On Reliable Broadcast in a Radio Network. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Principles of Distributed Computing*, PODC '05, pages 138–147, New York, NY, USA, 2005. ACM. doi:10.1145/1073814.1073841.

4 Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested Append-only Memory: Making Adversaries Stick to Their Word. *SIGOPS Oper. Syst. Rev.*, 41(6):189–204, October 2007. doi:10.1145/1323293.1294280.

5 Allen Clement, Flavio Junqueira, Aniket Kate, and Rodrigo Rodrigues. On the (Limited) Power of Non-equivocation. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing*, PODC '12, pages 301–308, New York, NY, USA, 2012. ACM. doi:10.1145/2332432.2332490.

6 Jeffrey Considine, Matthias Fitzi, Matthew Franklin, Leonid A. Levin, Ueli Maurer, and David Metcalf. Byzantine Agreement Given Partial Broadcast. *Journal of Cryptology*, 18(3):191–217, July 2005. doi:10.1007/s00145-005-0308-x.

7 Danny Dolev. The Byzantine generals strike again. *Journal of Algorithms*, 3(1):14–30, 1982. doi:10.1016/0196-6774(82)90004-9.

8 Michael J. Fischer, Nancy A. Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1(1):26–39, March 1986. doi:10.1007/BF01843568.

9 Mattias Fitzi and Ueli Maurer. From Partial Consistency to Global Broadcast. In *Proceedings of the Thirty-second Annual ACM Symposium on Theory of Computing*, STOC '00, pages 494–503, New York, NY, USA, 2000. ACM. doi:10.1145/335305.335363.

10 M. Franklin and M. Yung. Secure Hypergraphs: Privacy from Partial Broadcast. *SIAM Journal on Discrete Mathematics*, 18(3):437–450, 2004. doi:10.1137/S0895480198335215.

11 Matthew Franklin and Rebecca N. Wright. Secure Communication in Minimal Connectivity Models. *Journal of Cryptology*, 13(1):9–30, January 2000. doi:10.1007/s001459910002.

12 Alexander Jaffe, Thomas Moscibroda, and Siddhartha Sen. On the Price of Equivocation in Byzantine Agreement. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing*, PODC '12, pages 309–318, New York, NY, USA, 2012. ACM. doi:10.1145/2332432.2332491.

13 Muhammad Samir Khan, Syed Shalan Naqvi, and Nitin H. Vaidya. Exact Byzantine Consensus on Undirected Graphs Under Local Broadcast Model. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, pages 327–336, New York, NY, USA, 2019. ACM. doi:10.1145/3293611.3331619.

14 Muhammad Samir Khan, Lewis Tseng, and Nitin H. Vaidya. Exact Byzantine Consensus on Arbitrary Directed Graphs under Local Broadcast Model. *CoRR*, 2019. arXiv:1911.07298.

15 Chiu-Yuen Koo. Broadcast in Radio Networks Tolerating Byzantine Adversarial Behavior. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, PODC '04, pages 275–282, New York, NY, USA, 2004. ACM. doi:10.1145/1011767.1011807.

16 Chiu-Yuen Koo, Vartika Bhandari, Jonathan Katz, and Nitin H. Vaidya. Reliable Broadcast in Radio Networks: The Bounded Collision Case. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Principles of Distributed Computing*, PODC '06, pages 258–264, New York, NY, USA, 2006. ACM. doi:10.1145/1146381.1146420.

17 Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982. doi:10.1145/357172.357176.

18 H. J. LeBlanc, H. Zhang, X. Koutsoukos, and S. Sundaram. Resilient Asymptotic Consensus in Robust Networks. *IEEE Journal on Selected Areas in Communications*, 31(4):766–781, April 2013. doi:10.1109/JSAC.2013.130413.

19 C. Li, M. Hurfin, Y. Wang, and L. Yu. Towards a Restrained Use of Non-Equivocation for Achieving Iterative Approximate Byzantine Consensus. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 710–719, May 2016. doi:10.1109/IPDPS.2016.62.

20 Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

21 Syed Shalan Naqvi, Muhammad Samir Khan, and Nitin H. Vaidya. Exact Byzantine Consensus Under Local-Broadcast Model. *CoRR*, abs/1811.08535, 2018. arXiv:1811.08535.

22 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.

23 T. Rabin and M. Ben-Or. Verifiable Secret Sharing and Multiparty Protocols with Honest Majority. In *Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing*, STOC '89, pages 73–85, New York, NY, USA, 1989. ACM. doi:10.1145/73007.73014.

24 D. V. S. Ravikant, V. Muthuramakrishnan, V. Srikanth, K. Srinathan, and C. Pandu Rangan. On Byzantine Agreement over (2,3)-Uniform Hypergraphs. In Rachid Guerraoui, editor, *Distributed Computing*, pages 450–464, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

25 Lewis Tseng and Nitin Vaidya. Iterative Approximate Byzantine Consensus under a Generalized Fault Model. In Davide Frey, Michel Raynal, Saswati Sarkar, Rudrapatna K. Shyamasundar, and Prasun Sinha, editors, *Distributed Computing and Networking*, pages 72–86, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

26 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.

27 Nitin H. Vaidya. Iterative Byzantine Vector Consensus in Incomplete Graphs. In Mainak Chatterjee, Jian-nong Cao, Kishore Kothapalli, and Sergio Rajsbaum, editors, *Distributed Computing and Networking*, pages 14–28, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

28 Nitin H. Vaidya, Lewis Tseng, and Guanfeng Liang. Iterative Approximate Byzantine Consensus in Arbitrary Directed Graphs. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing*, PODC '12, pages 365–374, New York, NY, USA, 2012. ACM. doi:10.1145/2332432.2332505.

29 Yongge Wang and Yvo Desmedt. Secure Communication in Multicast Channels: The Answer to Franklin and Wright's Question. *Journal of Cryptology*, 14(2):121–135, March 2001. doi:10.1007/s00145-001-0002-y.

30 H. Zhang and S. Sundaram. Robustness of information diffusion algorithms to locally bounded adversaries. In *2012 American Control Conference (ACC)*, pages 5855–5861, June 2012. doi:10.1109/ACC.2012.6315661.

Reconfigurable Lattice Agreement and Applications

Petr Kuznetsov
LTCI, Télécom Paris, Institut Polytechnique Paris, Paris, France
petr.kuznetsov@telecom-paris.fr

Thibault Rieutord
CEA LIST, PC 174, Gif-sur-Yvette, 91191, France
thibault.rieutord@cea.fr

Sara Tucci-Piergiovanni
CEA LIST, PC 174, Gif-sur-Yvette, 91191, France
sara.tucci@cea.fr

Abstract

Reconfiguration is one of the central mechanisms in distributed systems. Due to failures and connectivity disruptions, the very set of service replicas (or *servers*) and their roles in the computation may have to be reconfigured over time. To provide the desired level of consistency and availability to applications running on top of these servers, the *clients* of the service should be able to reach some form of agreement on the system configuration. We observe that this agreement is naturally captured via a *lattice* partial order on the system states. We propose an asynchronous implementation of *reconfigurable* lattice agreement that implies elegant reconfigurable versions of a large class of *lattice* abstract data types, such as max-registers and conflict detectors, as well as popular distributed programming abstractions, such as atomic snapshot and commit-adopt.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Reconfigurable services, lattice agreement

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.31

Related Version A full version of the paper is available at `https://arxiv.org/abs/1910.09264`.

1 Introduction

A decentralized service [6, 14, 25, 28] runs on a set of fault-prone *servers* that store replicas of the system state and run a synchronization protocol to ensure consistency of concurrent data accesses. In the context of a storage system exporting read and write operations, several proposals [2,3,18,20,23,31] came out with a reconfiguration interface that allows the servers to join and leave while ensuring consistency of the stored data. Early proposals of reconfigurable storage systems [20] were based on using *consensus* [16, 21] to ensure that replicas *agree* on the evolution of the system membership. Consensus, however, is expensive and difficult to implement, and recent solutions [2, 3, 18, 23, 31] replace consensus with weaker abstractions capturing the minimal coordination required to safely modify the system configuration. These solutions, however, lack a uniform way of deriving reconfigurable versions of static objects.

Lattice objects. In this paper, we propose a universal reconfigurable construction for a large class of objects. Unlike a consensus-based reconfiguration proposed earlier for generic state-machine replication [26], our construction is purely asynchronous, at the expense of assuming a restricted object behavior. More precisely, we assume that the set $\mathcal{L}$ of the object's states can be represented as a (join semi-) *lattice* $(\mathcal{L}, \sqsubseteq)$, where $\mathcal{L}$ is partially ordered by the binary relation $\sqsubseteq$ such that for all elements of $x, y \in \mathcal{L}$, there exists the *least upper bound* in $\mathcal{L}$, denoted $x \sqcup y$, where $\sqcup$, called the *join* operator, is an associative, commutative,

© Petr Kuznetsov, Thibault Rieutord, and Sara Tucci-Piergiovanni;
licensed under Creative Commons License CC-BY
23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller; Article No. 31; pp. 31:1–31:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

and idempotent binary operator on $\mathcal{L}$. Many important data types, such as sets and counters, as well as useful concurrent abstractions, such as conflict detector [5], can be expressed this way. Intuitively, $x \sqcup y$ can be seen as a *merge* of two alternatively proposed updated states x and y. Thus, an implementation ensuring that all "observable" states are ordered by $\sqsubseteq$ cannot be distinguished from an atomic object.

Consider, for example, the *max-register* [4] data type with two operations: *writeMax* writes a value and *readMax* returns the largest value written so far. Its state space can be represented as a lattice $(\sqsubseteq, \sqcup)$ of its values, where $\sqsubseteq = \leq$ and $x \sqcup y = \max(x, y)$. Intuitively, a linearizable implementation of max-register must ensure that every read value is a join of previously proposed values, and all read values are totally ordered (with respect to $\leq$).

Reconfigurable lattice agreement. In this paper, we introduce the *reconfigurable lattice agreement* [8,15]. It is natural to treat the *system configuration*, i.e., the set of servers available for data replication, as an element in a lattice. A lattice-defined join of configurations, possibly concurrently proposed by different clients, results in a new configuration. The lattice-agreement protocol ensures that configurations evaluated by concurrent processes are *ordered*. Despite processes possibly disagreeing about the precise configuration they belong to, they can use the configurations relative ordering to maintain the system data consistency.

A configuration is defined by a set of servers, a quorum system [19], i.e., a set system ensuring the intersection property[1] and, possibly, other parameters. For example, elements of a reconfiguration lattice can be defined as sets of *configuration updates*: each such update either adds a server to the configuration or removes a server from it. The *members* of such a configuration are the set of all servers that were added but not yet removed. A join of two configurations defined this way is simply a union of their updates (this approach is implicitly used in earlier asynchronous reconfigurable constructions [2, 18, 31]).

Reconfigurable L-ADTs and applications. We show that our reconfigurable lattice agreement, defined on a product of a *configuration lattice* and an *object lattice*, immediately implies reconfigurable versions of many sequential types, such as *max-register* and *conflict detector*. More generally, any *state-based commutative* abstract data (called *L-ADT*, for *lattice abstract data type*, in this paper) has a reconfigurable *interval-linearizable* [12] implementation. Intuitively, interval-linearizability [12], a generalization of the classical linearizability [22], allows specifying the behavior of an object when multiple concurrent operations "influence" each other. Their effects are then merged using a join operator, which turns out to be natural in the context of reconfigurable objects.

Our transformations are straightforward. To get an (interval-linearizable) reconfigurable implementation of an L-ADT, we simply use its state lattice, as a parameter, in our reconfigurable lattice agreement. The resulting implementations are naturally composable: we get a reconfigurable composition of two L-ADTs by using a *product* of their lattices. If operations on the object can be partitioned into *updates* (modifying the object state without providing informative responses) and *queries* (not modifying the object state), as in the case of max-registers, the reconfigurable implementation becomes linearizable[2].

---

[1] The most commonly used quorum system is majority-based: quorums are all majorities of servers. We can, however, use any other quorum system, as suggested in [20, 23].

[2] Such "update-query" L-ADTs are known as state-based convergent replicated data types (CvRDT) [29]. These include *max-register*, *set* and *abort flag* (a new type introduced in this paper).

We then use our reconfigurable implementations of *max-register*, *conflict detector*, *set* and *abort-flag* to devise reconfigurable versions of *atomic snapshot* [1], *commit-adopt* [17] and *safe agreement* [10]. Figure 1 shows how are constructions are related. Due to lack of space, the *safe agreement* and its associated objects are delegated to the technical report [24].

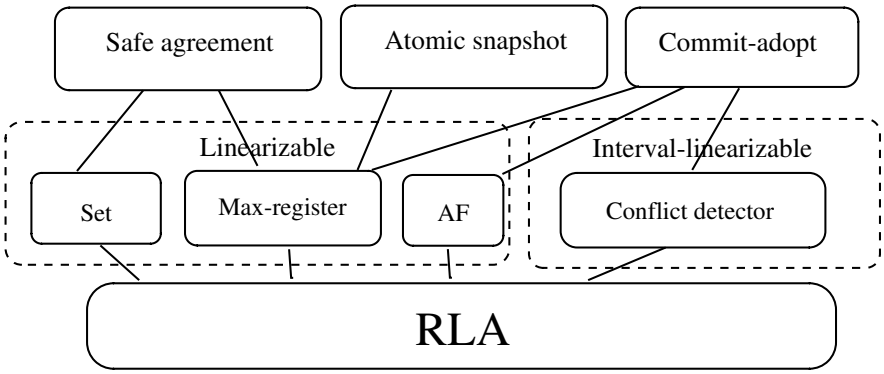


Figure 1 Our reconfigurable implementations: reconfigurable lattice agreement (RLA) is used to construct linearizable implementations of a set, a max-register, an abort flag, and an interval-linearizable implementation of a conflict detector. On top of max-registers we construct an atomic snapshot; on top of a max-register, an abort-flag, and a conflict detector, we construct a commit-adopt abstraction; and, on top of sets and a max register, we implement a safe agreement abstraction.

Summary. Our reconfigurable construction is the first to be, at the same time:

- Asynchronous, unlike consensus-based solutions [13, 20, 26], and not assuming an external lattice agreement service [23];
- Uniformly applicable to a large class of objects, unlike existing reconfigurable systems that either focus on read-write storage [2,18,20,23] or require data type-specific implementations of exported reconfiguration interfaces [31];
- Allowing for a straightforward composition of reconfigurable objects;
- Maintaining configurations with abstract *quorum systems* [19], not restricted to *majority-based* quorums [2, 18];
- Exhibiting optimal time complexity and message complexity comparable with the best known implementations [2, 23, 31];
- Logically separating *clients* (external entities that use the implemented service) from *servers* (entities that maintain the service and can be reconfigured).

We also believe our reconfigurable construction to be the simplest on the market, using only twenty one lines of pseudocode and provided with a concise proof.

Roadmap. The rest of the paper is organized as follows. We give basic model definitions in Section 2. In Section 3, we define our type of reconfigurable objects, followed by the related notion of reconfigurable lattice agreement in Section 4. In Section 5, we describe our implementation of reconfigurable lattice agreement, and, in Section 6, we show how to use it to implement a reconfigurable L-ADT object. In Section 7 we describe some possible applications. We conclude with, in Section 8, an overview of the related work, and, in Section 9, a discussion on algorithms complexity and possible trade-offs. The full version of this paper, with detailed proofs, is available at [24].

2 Definitions

Replicas and clients. Let Π be a (possibly infinite) set of potentially participating processes. A subset of the processes, called *replicas*, are used to maintain a replicated object. A process can also act as a *client*, invoking operations on the object and proposing system reconfigurations. Both replicas and clients are subject to crash failures: a process *fails* when it prematurely stops taking steps of its algorithm. A *failure model* stipulates when and where failures might occur. We present our failure model in Section 4, where we formally define reconfigurable lattice agreement.

Abstract data types. An abstract data type (ADT) is a tuple $T = (A, B, Z, z_0, \tau, \delta)$. Here A and B are countable sets called the *inputs* and *outputs*. Z is a countable set of abstract object *states*, $z_0 \in Z$ being the initial state of the object. The map $\tau : Z \times A \to Z$ is the *transition function*, specifying the effect of an input on the object state and the map $\delta : Z \times A \to B$ is the *output function*, specifying the output returned for a given input and object local state. The input represents an operation with its parameters, where (i) the operation can have a side-effect that changes the abstract state according to transition function τ and (ii) the operation can return values taken in the output B, which depends on the state in which it is called and the output function δ (for simplicity, we only consider deterministic types here, check, e.g., [27], for more details.)

Interval linearizability. We now briefly recall the notion of *interval-linearizability* [12], a recent generalization of linearizability [22].

Let us consider an abstract data type $T = (A, B, Z, z_0, \tau, \delta)$. A *history* of T is a sequence of inputs (elements of A) and outputs (elements of B), each labeled with a process identifier and an operation identifier. An *interval-sequential history* is a sequence:

$$z_0, I_1, R_1, z_1, I_2, R_2, z_2 \ldots, I_m, R_m, z_m,$$

where each $z_i \in Z$ is a state, $I_i \subseteq A$ is a set of inputs, and $R_i \subseteq B$ is a set of outputs. An *interval-sequential specification* is a set of interval-sequential histories.

We only consider *well-formed* histories. Informally, in a well-formed history, a process only invokes an operation once its previous operation has returned and every response r is preceded by a "matching" operation i.

A history H is *interval-linearizable* respectively to an interval-sequential specification $\mathcal{S}$ if it can be *completed* (by adding matching responses to incomplete operations) so that the resulting history $\bar{H}$ can be associated with an interval-sequential history S such that: (1) $\bar{H}$ and S are *equivalent*, i.e., $\forall p \in \Pi$, $\bar{H}|p = S|p$, (2) $S \in \mathcal{S}$, and (3) $\to_H \subseteq \to_S$, i.e., S preserves the real-time precedence relation of H. (Check [12] for more details on the definition.)

Lattice agreement. An abstract (join semi-)*lattice* is a tuple $(\mathcal{L}, \sqsubseteq)$, where $\mathcal{L}$ is a set partially ordered by the binary relation $\sqsubseteq$ such that for all elements of $x, y \in \mathcal{L}$, there exists the least upper bound for the set $\{x, y\}$. The least upper bound is an associative, commutative, and idempotent binary operation on $\mathcal{L}$, denoted by $\sqcup$ and called the *join operator* on $\mathcal{L}$. We write $x \sqsubset y$ whenever $x \sqsubseteq y$ and $x \neq y$. With a slight abuse of notation, for a set $L \subseteq \mathcal{L}$, we also write $\bigsqcup L$ for $\bigsqcup_{x \in L} x$, i.e., $\bigsqcup L$ is the join of the elements of L.

Notice that two lattices $(\mathcal{L}_1, \sqsubseteq_1)$ and $(\mathcal{L}_2, \sqsubseteq_2)$ naturally imply a *product* lattice $(\mathcal{L}_1 \times \mathcal{L}_2, \sqsubseteq_1 \times \sqsubseteq_2)$ with a product join operator $\sqcup = \sqcup_1 \times \sqcup_2$. Here for all $(x_1, x_2), (y_1, y_2) \in \mathcal{L}_1 \times \mathcal{L}_2$, $(x_1, x_2)(\sqsubseteq_1 \times \sqsubseteq_2)(y_1, y_2)$ if and only if $x_1 \sqsubseteq_1 y_1$ and $x_2 \sqsubseteq_2 y_2$.

The (generalized) *lattice agreement* concurrent abstraction, defined on a lattice $(\mathcal{L}, \sqsubseteq)$, exports a single operation *propose* that takes an element of $\mathcal{L}$ as an argument and returns an element of $\mathcal{L}$ as a response. When the operation *propose*(x) is invoked by process p we say that p *proposes* v, and when the operation returns v' we say that p *learns* v'. Assuming that no process invokes a new operation before its previous operation returns, the abstraction satisfies the following properties:

- **Validity.** If a *propose*(v) operation returns a value v' then v' is a join of some proposed values including v and all values learnt before the invocation of the operation.
- **Consistency.** The learnt values are totally ordered by $\sqsubseteq$.
- **Liveness.** If a process invokes a *propose* operation and does not fail then the operation eventually returns.

A historical remark. The original definition of long-lived lattice agreement [15] separates "receive" events and "learn" events. Here we suggest a simpler definition that represents the two events as the invocation and the response of a *propose* operation. This also allows us to slightly strengthen the validity condition so that it accounts for the *precedence* relation between *propose* operations. As a result, we can directly relate lattice agreement to linearizable [22] and interval-linearizable [12] implementations, without introducing artificial "nop" operations [15].

3 Lattice Abstract Data Type

In this section, we introduce a class of types that we call *lattice abstract data types* or *L-ADT*. In an *L-ADT*, the set of states forms a join semi-lattice with a partial order $\sqsubseteq^Z$. A lattice object is therefore defined as a tuple $L = (A, B, (Z, \sqsubseteq^Z, \sqcup^Z), z_0, \tau, \delta)$.[3] Moreover, the transition function δ must comply with the partial order $\sqsubseteq^Z$, that is $\forall z, a \in Z \times A : z \sqsubseteq^Z \tau(z, a)$, and the composition of transitions must comply with the join operator, that is $\forall z \in Z, \forall a, a' \in A : \tau(\tau(z, a), a') = \tau(z, a) \sqcup^Z \tau(z, a') = \tau(\tau(z, a'), a)$. Hence, we can say that the transition function is "commutative".

Update-query L-ADT. We say an L-ADT $L = (A, B, (Z, \sqsubseteq^Z, \sqcup^Z), z_0, \tau, \delta)$ is *update-query* if A can be partitioned in *updates* U and *queries* Q such that:

- there exists a special "dummy" response $\perp$ (z_0 may also be used) such that $\forall u \in U,\ z \in Z$, $\delta(u, z) = \perp$, i.e., updates do not return informative responses;
- $\forall q \in Q,\ z \in Z,\ \tau(q, z) = z$, i.e., queries do not modify the states.

This class is also known as a state-based convergent replicated data types (CvRDT) [29]. Typical examples of update-query L-ADTs are *max-register* [4] (see Section 1) or *sets*. Note that any (L-)ADT can be transformed into an update-query (L-)ADT by "splitting its operations" into an update and a query (see [27]).

Composition of L-ADTs. The composition of two ADTs $T = (A, B, Z, z_0, \tau, \delta)$ and $T' = (A', B', Z', z'_0, \tau', \delta')$ is denoted $T \times T'$ and is equal to $(A + A', B \cup B', Z \times Z', (z_0, z'_0), \tau'', \delta'')$; where $A + A'$ denotes the disjoint union and where τ'' and δ'' apply, according to the domain A or A' of the input, either τ and δ or τ' and δ' on their respecting half of the state (see [27]).

Since the cartesian product of two lattices remains a lattice, the composition of L-ADTs is naturally defined and produces an L-ADT. The composition is also closed to update-query ADT, and thus to update-query L-ADT. Moreover, the composition is an associative and commutative operator, and hence, can easily be used to construct elaborate L-ADT.

[3] For convenience, we explicitly specify the join operator $\sqcup^Z$ here, i.e., the least upper bound of $\sqsubseteq^Z$.

Configurations as L-ADTs. Let us also use the formlism of L-ADT to define a *configuration L-ADT* as a tuple $(A^{\mathcal{C}}, B^{\mathcal{C}}, (\mathcal{C}, \sqsubseteq^{\mathcal{C}}, \sqcup^{\mathcal{C}}), C_0, \tau^{\mathcal{C}}, \delta^{\mathcal{C}})$ with $C_0 \in \mathcal{C}$ the *initial configuration*. For each element C of the *configuration lattice* $\mathcal{C}$, the input set A includes the query operations $members()$, such that $\delta^{\mathcal{C}}(C, members()) \subseteq \Pi$, and $quorums()$ where $\delta^{\mathcal{C}}(C, quorums()) \subseteq 2^{\delta^{\mathcal{C}}(C, members())}$ is a *quorum system*, that is, every two subsets in $\delta^{\mathcal{C}}(C, quorums())$ have a non-empty intersection. With a slight abuse of notation, we will write these operations as $members(C)$ and $quorums(C)$.

For example, $\mathcal{C}$ can be the set of tuples (In, Out), where $In \subseteq \Pi$ is a set of *activated* processes, and $Out \subseteq \Pi$ is a set of *removed* processes. Then $\sqsubseteq^{\mathcal{C}}$ can be defined as the piecewise set inclusion on (In, Out). The set of members of (In, Out) will simply be $In - Out$ and the set of quorums (pairwise-intersecting subsets of $In - Out$), e.g., all majorities of $In - Out$. Operations in $A^{\mathcal{C}}$ can be $add(s)$, $s \in \Pi$, that adds s to the set of activated processes and $remove(s)$, $s \in \Pi$, that adds s to the set of removed processes of a configuration. One can easily see that updates "commute" and that the type is indeed a configuration L-ADT. Let us note that L-ADTs allow for more expressive reconfiguration operations than simple *adds* and *removes*, e.g., maintaining a minimal number of members in a configuration or adapting the quorum system dynamically, as studied in detail by Jehl et al. in [23].

Interval-sequential specifications of L-ADTs. Let $L = (A, B, (Z, \sqsubseteq^Z, \sqcup^Z), z_0, \tau, \delta)$ be an L-ADT. As τ "commutes", the state reached after a sequence of transitions is order-independent. Hence, we can define a natural, deterministic, interval-sequential specification of L, $\mathcal{S}_L$, as the set of interval-sequential histories $z_0, I_1, R_1, z_1, \ldots, I_m, R_m, z_m$ such that:

- $\forall i = 1, \ldots, m$, $z_i = \bigsqcup^Z_{a \in I_{i-1}} \tau(a, z_{i-1})$, i.e., every state z_i is a join of operations in I_{i-1} applied to z_{i-1}.
- $\forall i = 1, \ldots, m$, $\forall r \in R_i$, $r = \delta(a, z_i)$, where a is the matching invocation operation for r, i.e., every response in R_i is the result of the associated operation applied to state z_i.

4 Reconfigurable lattice agreement: definition

We define a reconfigurable lattice $(\mathcal{L}, \sqsubseteq)$ as the product of the state spaces of an *object* L-ADT $(A^{\mathcal{O}}, B^{\mathcal{O}}, (\mathcal{O}, \sqsubseteq^{\mathcal{O}}, \sqcup^{\mathcal{O}}), O_0, \tau^{\mathcal{O}}, \delta^{\mathcal{O}})$ and a *configuration* L-ADT $(A^{\mathcal{C}}, B^{\mathcal{C}}, (\mathcal{C}, \sqsubseteq^{\mathcal{C}}, \sqcup^{\mathcal{C}}), C_0, \tau^{\mathcal{C}}, \delta^{\mathcal{C}})$ (see Section 3). That is, $(\mathcal{L}, \sqsubseteq) = (\mathcal{O} \times \mathcal{C}, \sqsubseteq^{\mathcal{O}} \times \sqsubseteq^{\mathcal{C}})$ with the product join operator $\sqcup = \sqcup^{\mathcal{O}} \times \sqcup^{\mathcal{C}}$. Our main tool is the reconfigurable lattice agreement, a generalization of lattice agreement operating on $(\mathcal{L}, \sqsubseteq)$. We say that $\mathcal{L}$ is the set of *states*. For a state $u = (O, C) \in \mathcal{L}$, we use notations $u.O = O$ and $u.C = C$.

Failure model. When a client p invokes $propose((O, C))$, we say that p *proposes* object state O and configuration state C. We say that p *learns* an object state O' and a configuration C' if its *propose* invocation returns (O', C').

We say that a configuration C is *potential* if there is a set $\{C_1, \ldots, C_k\}$ of proposed configurations such that $C = C_0 \sqcup^{\mathcal{C}} (\bigsqcup^{\mathcal{C}}_{i=1,\ldots,k} C_i)$ (with C_0 the initial configuration). A configuration C is said to be *superseded* as soon as a process learns a state $(*, C')$ with $C \sqsubseteq^{\mathcal{C}} C'$ and $C \neq C'$. At any moment of time, a configuration is *active* if it is a potential but not yet superseded configuration. Intuitively, some quorum of a configuration should remain "reachable" as long as the configuration is active.

We say that a replica r is *active* when it is a member of an active configuration C, i.e., $r \in member(C)$. A replica is *correct* if, from some point on, it is forever active and not failed. A *client* is correct if it does not fail while executing a *propose* operation.

A configuration C is *available* if some set of replicas in $quorums(C)$ contains only correct processes. In arguing liveness in this paper, we assume the following:

- **Configuration availability.** Any potential configuration that is never superseded must be available.

Therefore, if a configuration is superseded by a strictly larger (w.r.t. $\sqsubseteq^{\mathcal{C}}$) one, then it does not have to be available, i.e., we can safely remove some replicas from it for maintenance.

Liveness properties. In a constantly reconfigured system, we may not be able to ensure liveness to all operations. A slow client can be always behind the active configurations: its set of estimated potential configurations can always be found to constitute a superseded configuration. Therefore, for liveness, we assume that only finitely many reconfigurations occur. Otherwise, only lock-freedom may be provided.

Therefore, to get a reconfigurable object, we replace the liveness property of lattice agreement with the following one:

- **Reconfigurable Liveness.** In executions with finitely many distinct proposed configurations, every *propose* operation invoked by a correct client eventually returns.

Thus, the desired liveness guarantees are ensured as long as only finitely many distinct configurations are proposed. However, the clients are free to perform infinitely many *object* updates without making any correct client starve.

Formally, *reconfigurable lattice agreement* defined on $(\mathcal{L}, \sqsubseteq) = (\mathcal{O} \times \mathcal{C}, \sqsubseteq^{\mathcal{O}} \times \sqsubseteq^{\mathcal{C}})$ satisfies the Validity and Consistency properties of lattice agreement (see Section 2) and the Reconfigurable Liveness property above.

Furthermore, we can only guarantee liveness to clients assuming that, eventually, every correct system participant (client or replica) is informed of the currently active configuration. It boils down to ensuring that an eventually consistent reconfigurable memory is available to store the greatest learnt configuration.

For simplicity, we assume that a reliable broadcast primitive [11] is available, ensuring that (i) every broadcast message was previously broadcast, (ii) if a correct process broadcasts a message m, then it eventually delivers m, and (iii) every message delivered by a correct process is eventually delivered by every correct process. Note that *Configuration availability* implies that an active configuration is either available or sufficiently responsive to be superseded.

5 Reconfigurable lattice agreement: implementation

We now present our main technical result, reconfigurable (generalized) lattice agreement. This algorithm will then be used to implement reconfigurable objects.

Overview. The algorithm for every process p (client or replica) is presented in Algorithm 1. We assume that all procedures (including sub-calls to the *updateState* procedure) are executed by p *sequentially* until they terminate or get interrupted by the wait condition in line 9.

Every process (client or server) p maintains a *state* variable $v_p \in \mathcal{L}$ storing its local estimate of the greatest committed object ($v_p.O$) and configuration ($v_p.C$) states, initialized to the initial element of the lattice (O_0, C_0). We say that a state is *committed* if a process broadcasts it in line 13. Note that all learnt states are previously committed (either directly by the learning process or indirectly by another process). Every process p also maintains T_p, the set of *pending input* configuration states, i.e., known input configuration states that are not superseded by the committed state estimate v_p. For the object lattice, processes stores in obj_p the join of all known proposed objects states.

Algorithm 1 Reconfigurable universal construction: code for process p.

Local variables:
seq_p, initially 0 {*The number of issued requests* }
v_p, initially (O_0, C_0) {*The last learnt state* }
T_p, initially $\emptyset$ {*The set of proposed configuration states* }
obj_p, initially O_0 {*The candidate object state* }

operation *propose*(*prop*) {*Propose a new state prop* } {*Executed by a client* }
1 $updateState(v_p, prop.O, \{prop.C\})$
2 $learnLB := \bot$
3 **while** *true* **do**
4 $seq_p := seq_p + 1$
5 $oldCommit := v_p$ {*Archive commit estimate* }
6 $oldCandidates := (obj_p, T_p)$ {*Archive candidate states* }
7 $V := \{\bigsqcup^{\mathcal{C}}(\{v_p.C\} \cup S) \mid S \subseteq T_p\}$ {*Queried configurations* }
8 send $\langle (REQ, seq_p), (v_p, obj_p, T_p)\rangle$ to $\bigcup_{u \in V} members(u)$
9 **wait until** $oldCommit.C \neq v_p.C$ or $\forall u \in V$, received responses of the type $\langle (RESP, seq_p), _\rangle$ from some $Q \in quorums(u)$
10 **if** $oldCommit.C = v_p.C \wedge oldCandidates = (_, T_p)$ **then** {*Stable configurations* }
11 **if** $learnLB = \bot$ **then** $learnLB = (obj_p, \bigsqcup^{\mathcal{C}}(\{v_p.C\} \cup T_p)$
12 **if** $oldCandidates = (obj_p, _)$ **then** {*No greater object received* }
13 *broadcast* $\langle COMMIT, (obj_p, \bigsqcup^{\mathcal{C}}(\{v_p.C\} \cup T_p))\rangle$
14 **return** $(obj_p, \bigsqcup^{\mathcal{C}}(\{v_p.C\} \cup T_p))$
15 **if** $learnLB \neq \bot \wedge learnLB \sqsubseteq v_p$ **then return** v_p {*Adopt learnt state* }

upon *receive* $\langle msgType, msgContent\rangle$ from process q
16 $updateState(msgContent)$ {*Update tracked states* }
17 **if** $msgType = (REQ, seq)$ **then** send $\langle (RESP, seq), (v_p, obj_p, T_p)\rangle$ to q

upon *deliver* $\langle COMMIT, state\rangle$
18 $updateState(state, state.O, \emptyset)$ {*Adopt the committed state* }

procedure $updateState(v, s_O, S_C)$ {*Merge tracked states* }
19 $v_p := v_p \sqcup v$ {*Update the commit estimate* }
20 $obj_p := obj_p \sqcup^{\mathcal{O}} s_O$ {*Update the object candidate* }
21 $T_p := \{u \in (T_p \cup S_C) \mid u \not\sqsubseteq^{\mathcal{C}} v_p.C\}$ {*Update and trim input candidates* }

To propose a new state, *prop*, client p updates its local variables through the *updateState* procedure using its input object and configuration states, $prop.O$ and $prop.C$ (line 1). The client p then enters a while loop where it sends a *request* containing (v_p, obj_p, T_p) and equipped with its current sequence number seq_p to all replicas from *every possible join* of pending input configurations with the commit estimate configuration state (line 7). Then p waits until either (1) a greater committed configuration is discovered from the quorum responses or the underlying reliable broadcast (line 18), or (2) a quorum of members of every reached configuration responds with messages of the type $\langle (RESP, seq_p), (v, s_O, S_C)\rangle$, where (v, s_O, S_C) corresponds to the replica updated triple (v_p, obj_p, T_p) (lines 8–9).

When a process (client or replica) p receives a new request or response of the type $\langle msgType, (v, s_O, S_C)\rangle$ or delivers a new committed state (line 18), it updates its commit estimate and its object candidate by joining its current values with the one received or delivered. The process also merges its set of pending input configurations T_p with the

received pending input configurations, except for those superseded by the updated commit estimate (lines 19–21). Every replica also sends a response containing the updated triple (v_p, obj_p, T_p) to the sender of the request (line 17).

If responses from quorums of all accessed configurations are received and no response contains a *new* (not yet known) pending input configuration or a *greater* object state, then the couple formed by obj_p and the join of the commit estimate configuration with all pending input configurations, i.e., $\bigsqcup^{\mathcal{C}}(\{v.C\} \cup T_p)$, is broadcasted and then returned as the new learnt state (lines 12-14). Otherwise, the client proceeds to a new round of the while loop.

To ensure wait-freedom, we integrate a *helping mechanism* consisting of having the clients adopt their committed state estimate (line 15). But, to know when a committed state can be returned, the client must first complete a communication round without interference from reconfigurations (line 11). After such a round, a committed state greater than the join of all known states, stored in *learnLB*, can safely be returned.

Correctness. We give here a short intuition of why our algorithm indeed implements reconfigurable lattice agreement (we refer the reader to [24] for a complete proof). We show first that the *updateState* function is actually a *join* operator on a lattice defined on the triples (v_p, obj_p, T_p), and that the join operator is consistent with the lattice partial order on $\mathcal{L}$ for the projection $(obj_p, \bigsqcup^{\mathcal{C}}(\{v_p.C\} \cup T_p))$. This way we can see our algorithm as the classical (generalized) lattice agreement [15] where the proposed elements are joined until no incompatibles values are received. However, there are some additional subtleties here.

First, for the validity property, we require that every learnt state includes all states that were learnt before the corresponding *propose* operation started. Proving this property is similar to proving the consistency property, i.e., that all learnt states are totally ordered by $\sqsubseteq$. In both cases, we need to show that all committed states are transitively related to each other by $\sqsubseteq$ and the real-time partial order (precedence ordering). Intuitively, we get this by showing that two clients proposing operations either went through quorums of the same configuration or that one client was already aware of a committed state greater than the other client's committed state. This transitive property is the base of the safety proof.

Finally, to argue reconfigurable liveness, we show first that the algorithm is "lock-free", i.e., at least one correct client is making progress, and then show that this client helps other clients to complete their operations. Lock-freedom relies on the underlying broadcast mechanism and the property of *configuration availability* (Section 4): no correct client may be indefinitely blocked waiting for responses to a request. The broadcast mechanism ensures that correct clients are eventually aware of the greatest state committed so far. Hence, as finitely many distinct configurations are proposed, eventually all correct clients will query the same configurations, which must be available. Hence, all states proposed by a correct process will eventually be included in all learnt states and thus a non-committing correct process cannot starve as it will eventually adopt a committed state.

▶ **Theorem 1.** *Algorithm 1 implements reconfigurable lattice agreement.*

6 Reconfigurable objects

In this section, we use our reconfigurable lattice agreement (RLA) abstraction to construct an interval-linearizable reconfigurable implementation of any L-ADT L. The proofs of this section are relatively straightforward and can be found in the technical report [24].

6.1 Defining and implementing reconfigurable L-ADTs

Let us consider two L-ADTs, an *object* L-ADT $L^{\mathcal{O}} = (A^{\mathcal{O}}, B^{\mathcal{O}}, (\mathcal{O}, \sqsubseteq^{\mathcal{O}}, \sqcup^{\mathcal{O}}), O_0, \tau^{\mathcal{O}}, \delta^{\mathcal{O}})$ and a *configuration* L-ADT $L^{\mathcal{C}} = (A^{\mathcal{C}}, B^{\mathcal{C}}, (\mathcal{C}, \sqsubseteq^{\mathcal{C}}, \sqcup^{\mathcal{C}}), C_0, \tau^{\mathcal{C}}, \delta^{\mathcal{C}})$ (Section 2).

The corresponding *reconfigurable L-ADT* implementation, defined on the composition $L = L^{\mathcal{O}} \times L^{\mathcal{C}}$, exports operations in $A^{\mathcal{O}} \times A^{\mathcal{C}}$. It must be interval-linearizable (respectively to $\mathcal{S}_L$) and ensure Reconfigurable Liveness (under the configuration availability assumption).

In the reconfigurable implementation of L, presented in Algorithm 2, whenever a process invokes an operation $a \in A^{\mathcal{O}}$, it proposes a state, $\tau^{\mathcal{O}}(O_p, a)$ – the result from applying a to the last learnt state (initially, C_0) – to RLA, updates (O_p, C_p) and returns the response $\delta^{\mathcal{O}}(O_p, a)$ corresponding to the new learnt state. Similarly, to update the configuration, the process applies its operation to the last learnt configuration and proposes the resulting state to RLA.

Algorithm 2 Interval-linearizable implementation of L-ADT $L = L^{\mathcal{O}} \times L^{\mathcal{C}}$: code for process p.

Shared: *RLA*, reconfigurable lattice agreement
Local variables:
  O_p, initially O_0 {*The last learnt object state* }
  C_p, initially C_0 {*The last learnt configuration state* }
upon invocation of $a \in A^{\mathcal{O}}$ {*Object operation* }
1 $(O_p, C_p) := RLA.propose((\tau^{\mathcal{O}}(O_p, a), C_p))$
2 **return** $\delta^{\mathcal{O}}(O_p, a)$
upon invocation of $a \in A^{\mathcal{C}}$ {*Reconfiguration* }
3 $(O_p, C_p) := RLA.propose((O_p, \tau^{\mathcal{C}}(C_p, a)))$
4 **return** $\delta^{\mathcal{C}}(C_p, a)$

▶ **Theorem 2.** *Algorithm 2 is a reconfigurable implementation of an L-ADT.*

In the special case, when the L-ADT is *update-query*, the construction above produces a *linearizable* implementation:

▶ **Theorem 3.** *Algorithm 2 is a reconfigurable linearizable implementation of an update-query L-ADT.*

6.2 L-ADT examples

We provide four examples of L-ADTs that allow for interval-linearizable (Theorem 2) and linearizable (Theorem 3) reconfigurable implementations.

Max-register. The *max-register* sequential object defined on a totally ordered set $(V, \leq_V)$ provides operations $writeMax(v)$, $v \in V$, returning a default value $\perp$, and $readMax$ returning the largest value written so far (or $\perp$ if there are no preceding writes). We can define the type as an update-query L-ADT as follows:

$$MR_V = (writeMax(v)_{v \in V} \cup \{readMax\}, V \cup \{\perp\}, (V \cup \{\perp\}, \leq_V, max_V), \perp, \tau_{MR_V}, \delta_{MR_V}).$$

where $\leq_V$ is extended to $\perp$ with $\forall v \in V : \perp \leq_V v$, $\delta_{MR_V}(z, a) = z$ if $a = readMax$ and $\perp$ otherwise, and $\tau_{MR_V}(z, a) = max_V(z, v)$ if $a = writeMax(v)$ and z otherwise.

It is easy to see that $(V \cup \{\perp\}, \leq_V, max_V)$ is a join semi-lattice and the L-ADT MR_V satisfies the sequential *max-register* specification.

Set. The (add-only) *set* sequential object defined using a countable set V provides operations $addSet(v)$, $v \in V$, returning a default value $\bot$, and $readSet$ returning the set of all values added so far (or $\emptyset$ if there are no preceding add operation). We can define the type as an update-query L-ADT as follows:

$$Set_V = (addSet(v)_{v \in V} \cup \{readSet\}, 2^V \cup \{\bot\}, (2^V, \subseteq, \cup), \emptyset, \tau_{Set}, \delta_{Set}).$$

where $\subseteq$ and $\cup$ are the usual operators on sets, $\delta_{Set}(z, a) = z$ if $a = readSet$ and $\bot$ otherwise, and $\tau_{Set}(z, a) = z \cup \{v\}$ if $a = addSet(v)$ and z otherwise.

It is easy to see that $(2^V, \subseteq, \cup)$ is a join semi-lattice and the L-ADT Set_V satisfies the sequential (add-only) *set* specification.

Abort flag. An *abort-flag* object stores a boolean flag that can only be raised from $\bot$ to $\top$. Formally, the LADT AF is defined as follows:

$$AF = \big(\{abort, check\}, \{\bot, \top\}, (\{\bot, \top\}, \sqsubseteq^{AF}, \sqcup^{AF}), \bot, \tau_{AF}, \delta_{AF}\big)$$

where $\bot \sqsubseteq^{AF} \top$, $\tau_{AF}(z, abort) = \delta_{AF}(z, abort) = \top$, and $\tau_{AF}(z, check) = \delta_{AF}(z, check) = z$.

Conflict detector. The *conflict-detector* abstraction [5] exports operation $check(v)$, $v \in V$, that may return *true* ("conflict"), or *false* ("no conflict"). The abstraction respects the following properties:

- If no two *check* operations have different inputs, then no operation can return *true*.
- If two *check* operations have different inputs, then they cannot both return *false*.

A conflict detector can be specified as an L-ADT defined as follows:

$$CD = \big(check(v)_{v \in V}, \{true, false\}, (V \times \{\top, \bot\}, \sqsubseteq^{CD}, \sqcup^{CD}), \bot, \tau_{CD}, \delta_{CD}\big)$$

where

- $\bot \sqsubseteq^{CD} \top$; $\forall v \in V$, $\bot \sqsubseteq^{CD} v$ and $v \sqsubseteq^{CD} \top$; $\forall v, v' \in V$, $v \neq v' \Rightarrow v \not\sqsubseteq^{CD} v'$;
- $\tau_{CD}(z, check(v)) = v$ if $z = \bot$ or $z = v$, and $\tau_{CD}(z, check(v)) = \top$ otherwise;
- $\delta_{CD}(z, check(v)) = true$ if $z = \top$ and *false* otherwise.

Also, we can see that $v \sqcup^{CD} v' = v'$ if $v = v'$ or $v = \bot$, and $\top$ otherwise.

▶ **Theorem 4.** *Any interval-linearizable implementation of CD is a conflict detector.*

7 Applications

Many ADTs do not have commutative operations and, thus, do not belong to L-ADT. Moreover, many distributed programming abstractions do not have a sequential specification at all and, thus, cannot be defined as ADTs, needless to say as L-ADTs.

However, as we show, certain such objects can be implemented from L-ADT objects. As L-ADTs are naturally composable, the resulting implementations can be seen as using a single (composed) L-ADT object. By using a reconfigurable version of this L-ADT object, we obtain a reconfigurable implementation. In our constructions we omit talking about reconfigurations explicitly: to perform an operation on the configuration component of the system state, a process simply proposes it to the underlying RLA (see, e.g., Algorithm 2).

Our examples here are atomic snapshots [1] and commit-adopt [17]. The proofs of this section and a discussion of safe agreement [10] are delegated to the technical report [24].

Atomic snapshots

An m-position atomic-snapshot memory maintains an array of m positions and exports two operations, *update*(i, v), where $i \in \{1, \ldots, m\}$ is a location in the array and $v \in V$ – the value to be written, that returns a predefined value ok and *snapshot*() that returns an m-vector of elements in V. Its sequential specification stipulates that every *snapshot*() operation returns a vector that contains, in each index $i \in \{1, \ldots, m\}$, the value of the last preceding *update* operation on the ith position (or a predefined initial value, if there is no such *update*).

Registers using $MR_{\mathbb{N}\times V}$. We first consider the special case of a single register (1-position atomic snapshot). We describe its implementation from a *max-register*, assuming that the set of values V is totally-ordered with relation $\leq^V$. Let $\leq^{reg}$ be a total order on $\mathbb{N} \times V$ (defined lexicographically, first on $\leq$ and then, in case of equality, on $\leq^V$). Let *MR* be a max-register defined on $(\mathbb{N} \times V, \leq^{reg})$.

The idea is to associate each written value *val* with a *sequence number seq* and to store them in *MR* as a tuple (seq, val). To execute an operation *update*(v), the process first reads *MR* to get the "maximal" sequence number s written to *MR* so far. Then it writes $(s + 1, v)$ back to *MR*. Notice that multiple processes may use $s + 1$ in their *update* operations, but only for concurrent operations. Ties are then broken by choosing the maximal value in the second component in the tuple. A *snapshot* operation simply reads *MR* and returns the value in the tuple.

Using any reconfigurable linearizable implementation of *MR* (Theorem 3), we obtain a reconfigurable implementation of an atomic (linearizable) register. Intuitively, all values returned by *snapshot* (read) operations on *MR* can be totally ordered based on the corresponding sequence numbers (ties broken using $\leq^V$), which gives the order of *reads* in the corresponding sequential history S.

Atomic snapshots. Our implementation of an m-position atomic snapshot (depicted in Algorithm 3) is a straightforward generalization of the register implementation described above. Consider the L-ADT defined as the product of m max-register L-ADTs. In particular, the partial order of the L-ADT is the product of m (total) orders $\leq^{snap}$: $\leq^{reg_1} \times \cdots \times \leq^{reg_m}$.

We also enrich the interface of the type with a new query operation *readAll* that returns the vector of m values found in the m max-register components. Note that the resulting type is still an update-query L-ADT, and thus, by Theorem 3, we can use a reconfigurable linearizable implementation of this type, let us denote it by *MRset*.

To execute *update*(v, i) on the implemented atomic snapshot, a process performs a read on the ith component of *MRset* to get sequence number s of the returned tuple and perfroms *writeMax*$((s + 1, v))$ on the ith component. To execute a snapshot, the process performs *readAll* on *MRset* and returns the vector of the second element of each item of the array.

Similarly to the case of a single register, the results of all *snapshot* operations can be totally ordered using the $\leq^{snap}$ total order on the returned vectors. Placing the matching *update* operation accordingly, we get an equivalent sequential execution that respects the atomic snapshot specification.

▶ **Theorem 5.** *Algorithm 3 implements an m-component MWMR atomic snapshot.*

Algorithm 3 Simulation of an m-component atomic snapshot using an L-ADT.

```
operation update(i, v)        {update register i with v }
1   (s, −) := MRset[i].readMax
2   MRset[i].writeMax((s + 1, v))

operation snapshot()
3   r := MRset.readAll
4   return snap with ∀i ∈ {1, . . . , m}, r[i] = (−, snap[i])
```

The Commit-Adopt Abstraction

Let us take a more elaborated example, the commit-adopt abstraction [17]. It is defined through a single operation $propose(v)$, where v belongs to some input domain V. The operation returns a couple $(flag, v)$ with $v \in V$ and $flag \in \{commit, adopt\}$, so that the following conditions are satisfied:

- **Validity:** If a process returns $(_, v)$, then v is the input of some process.
- **Convergence:** If all inputs are v, then all outputs are $(commit, v)$.
- **Agreement:** If a process returns $(commit, v)$, then all outputs must be of type $(_, v)$.

We assume here that V, the set of values that can be proposed to the commit-adopt abstraction, is totally ordered. The assumption can be relaxed at the cost of a slightly more complicated algorithm (by replacing the max register with a set object for example).

Our implementation of (reconfigurable) commit-adopt uses a *conflict-detector* object CD (used to detect distinct proposals), a max-register MR_V (used to write non-conflicting proposals), and an *abort flag* object AF.

Our commit-adopt implementation is presented in Algorithm 4. In its *propose* operation, a process first accesses the *conflict-detector* object CD (line 1). Intuitively, the conflict detector makes sure that committing processes share a common proposal.

If the object returns *false* (no conflict detected), the process writes its proposal in the max-register MR_V (line 2) and then checks the abort flag AF. If the check operation returns $\perp$, then the proposed value is returned with the *commit* flag (line 4). Otherwise, the same value is returned with the *adopt* flag (line 3).

If a conflict is detected (CD returns *true*), then the process executes the *abort* operation on AF (line 6). Then the process reads the *max-register*. If a non-$\perp$ value is read (some value has been previously written to MR), the process adopts that value (line 9). Otherwise, the process adopts its own proposed value (line 8).

▶ **Theorem 6.** *Algorithm 4 implements commit-abort.*

8 Related Work

Lattice agreement. Attiya et al. [8] introduced the (one-shot) lattice agreement abstraction and, in the shared-memory context, described a wait-free reduction of lattice agreement to atomic snapshot. Falerio et al. [15] introduced the long-lived version of lattice agreement (adopted in this paper) and described an asynchronous message-passing implementation of lattice agreement assuming a majority of correct processes, with $\mathcal{O}(n)$ time complexity (in terms of message delays) in a system of n processes. Our RLA implementation in Section 5 builds upon this algorithm.

Algorithm 4 Commit-adopt implementation using L-ADTs.

```
operation propose(v)
    if CD.check(v) = false then        {check conflicts }
        MR_V.writeMax(v)
        if AF.check = ⊤ then return (adopt, v)        {adopt the input }
        else return (commit, v)        {commit proposal }
    else        {Try to abort in case of conflict }
        AF.abort        {raise abort flag }
        val := MR_V.readMax
        if val = ⊥ then return (adopt, v)        {adopt the input }
        else return (adopt, val)        {adopt the possibly committed value }
```

CRDT. Conflict-free replicated data types (CRDT) were introduced by Shapiro et al. [29] for eventually synchronous replicated services. The types are defined using the language of join semi-lattices and assume that type operations are partitioned in updates and queries. Falerio et al. [15] describe a "universal" construction of a linearizable CRDT from lattice agreement. Skrzypczak et al. [30] argue that avoiding consensus in such constructions may bring performance gains. In this paper, we consider a more general class of types (L-ADT) that are "state-commutative" but not necessarily "update-query" and leverage the recently introduced criterion of interval-linearizability [12] for *reconfigurable* implementations of L-ADTs using RLA.

Reconfiguration. *Passive reconfiguration* [7, 9] assumes that replicas enter and leave the system under an explicit *churn model*: if the churn assumptions are violated, consistency is not guaranteed. In the *active reconfiguration* model, processes explicitly propose configuration updates, e.g., sets of new process members. Early proposals, such as RAMBO [20] focused on read-write storage services and used consensus to ensure that the clients agree on the evolution of configurations.

Recent solutions [2, 3, 18, 23, 31] propose an asynchronous reconfiguration by replacing consensus with weaker abstractions capturing the minimal coordination required to safely modify the system configuration. Moreover, Freestore [3] proposes a modular solution to derive interchangeable consensus-based and asynchronous reconfiguration.

Asynchronous reconfiguration. Dynastore [2] was the first solution emulating a reconfigurable atomic read/write register without consensus: clients can asynchronously propose incremental additions or removals to the system configuration. Since proposals commute, concurrent proposals are collected together without the need of deciding on a total order. Assuming n proposals, a Dynastore client might, in the worst case, go through 2^{n-1} candidate configurations before converging to a final one. Assuming a run with a total number of configurations m, complexity is $\mathcal{O}(min(mn, 2^n))$.

SmartMerge [23] allows for reconfiguring not only the system membership but also its quorum system, excluding possible undesirable configurations. SmartMerge brings an interesting idea of using an external reconfiguration service based on lattice agreement [15], which allows us to reduce the number of traversed configurations to $O(n)$. However, this solution assumes that this "reconfiguration lattice" is always available and non-reconfigurable (as we showed in this paper, lattice agreement is a powerful tool that can itself be used to implement a large variety of objects).

Gafni and Malkhi [18] proposed the *parsimonious speculative snapshot* task based on the commit-adopt abstraction [17]. Reconfiguration, built on top of the proposed abstraction, has complexity $\mathcal{O}(n^2)$: n for the traversal and n for the complexity of the parsimonious speculative snapshot implementation. Spiegelman, Keidar and Malkhi [31] improved this work by proposing an optimal solution with time complexity $\mathcal{O}(n)$ by obtaining an amortized (per process) time complexity $\mathcal{O}(1)$ for speculative snapshots operations.

9 Concluding Remarks

To conclude, let us briefly discuss the complexity of our solution to the reconfiguration problem and give an overview of how our solution could be further extended.

Round-trip complexity. The main complexity metric considered in the literature is the maximal number of communication round-trips needed to complete a reconfiguration when c operations are concurrently proposed. In the worst case, each time a round of requests is completed in our algorithm, a new state is affecting T_p or obj_p, and hence we have at most c round-trips. Note that a round might be interrupted by receiving a greater committed state at most c times as committed states are totally ordered joins of proposed states. We are aware of only one other optimal solution with linear round-trip complexity, proposed Spiegelman et al. [31]. In their solution, the maximal number of round-trips is at least $4c$, twice more than ours. This has to do with the use of a shared memory simulation preventing to read and write at the same time and preventing from sending requests to distinct configurations in parallel. Moreover, they also use a similar interruption mechanism.

Querying multiple configurations at the same time might increase the round-trip delay as we need to wait for more responses. Still, we believe that when the number of requests scales with a constant factor, this impact is negligible.

Message complexity. As in earlier solutions, messages are of linear size in the number of distinct proposed configurations or collect operations on the implemented object.

The number of exchanged messages depends on the configuration lattice. With at most k members per configuration, each client may send at most $k * 2^n$ messages per round as there are, in the worst case, exponentially many configurations to query. But this upper bound may be reached only if joins of proposed configurations do not share replicas. We expect, however, that in most cases the concurrently proposed configurations have large overlaps: configuration updates are typically gradual. For example, when a configuration is defined as a set of updates (added and removed replicas), clients may send at most $k + \Delta * n$ requests per round, where Δ the number of replicas added per proposal. For small Δ, the total number of messages is of order k.

An interesting question is whether we can construct a composite complexity metric that combines the number of messages a process sends and the time it takes to complete a *propose* operation. Indeed, one may try to find dependencies between accessing few configurations sequentially versus accessing many configurations in parallel.

Complexity trade-offs. If the cost of querying many configurations in parallel outweigh the cost of contacting fewer configurations sequentially, we can use the approach from [31]. Intuitively, it boils down to solving an instance of generalized lattice agreement on the configurations and then querying the produced configurations, there can be $O(c)$ of them, where c is the number of concurrently proposed configurations.

Objects with "well-structured" its lattices can be implemented very efficiently. Take, for example, the totally ordered lattice of a max-register. In this case, processes can directly return the state stored in *LearnLB* in line 11. Indeed, not returning a committed state might only violate the *consistency* property. But if states are already totally ordered, then the *consistency* property always holds. Therefore, in the absence of reconfiguration calls, operations can return in a single round trip. It is in general interesting to investigate how the lattice structure might be leveraged.

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. Marcos Kawazoe Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic atomic storage without consensus. *J. ACM*, 58(2):7:1–7:32, 2011.
3. Eduardo Alchieri, Alysson Bessani, Fabíola Greve, and Joni da Silva Fraga. Efficient and Modular Consensus-Free Reconfiguration for Fault-Tolerant Storage. In *OPODIS*, pages 26:1–26:17, 2017.
4. James Aspnes, Hagit Attiya, and Keren Censor. Max Registers, Counters, and Monotone Circuits. In *PODC*, pages 36–45, 2009.
5. James Aspnes and Faith Ellen. Tight Bounds for Adopt-Commit Objects. *Theory Comput. Syst.*, 55(3):451–474, 2014. doi:10.1007/s00224-013-9448-1.
6. Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing Memory Robustly in Message Passing Systems. *Journal of the ACM (JACM)*, 42(2):124–142, 1995.
7. Hagit Attiya, Hyun Chul Chung, Faith Ellen, Saptaparni Kumar, and Jennifer L. Welch. Emulating a Shared Register in a System That Never Stops Changing. *IEEE Trans. Parallel Distrib. Syst.*, 30(3):544–559, 2019.
8. Hagit Attiya, Maurice Herlihy, and Ophir Rachman. Atomic Snapshots Using Lattice Agreement. *Distributed Comput.*, 8(3):121–132, 1995.
9. Roberto Baldoni, Silvia Bonomi, Anne-Marie Kermarrec, and Michel Raynal. Implementing a Register in a Dynamic Distributed System. In *ICDCS*, pages 639–647, 2009.
10. Elizabeth Borowsky and Eli Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *STOC*, pages 91–100, 1993.
11. Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to reliable and secure distributed programming.* Springer Science & Business Media, 2011.
12. Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. Unifying Concurrent Objects and Distributed Tasks: Interval-Linearizability. *J. ACM*, 65(6):45:1–45:42, 2018.
13. Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.
14. Gregory V. Chockler, Rachid Guerraoui, Idit Keidar, and Marko Vukolic. Reliable Distributed Storage. *IEEE Computer*, 42(4):60–67, 2009.
15. Jose Faleiro, Sriram Rajamani, Kaushik Rajan, Ganesan Ramalingam, and Kapil Vaswani. Generalized lattice agreement. In *PODC*, pages 125–134, 2012.
16. Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of Distributed Consensus with one Faulty Process. *jacm*, 32(2):374–382, 1985.
17. Eli Gafni. Round-by-round fault detectors: Unifying synchrony and asynchrony. In *PODC*, pages 143–152, 1998.
18. Eli Gafni and Dahlia Malkhi. Elastic Configuration Maintenance via a Parsimonious Speculating Snapshot Solution. In *DISC*, pages 140–153, 2015.
19. David K. Gifford. Weighted Voting for Replicated Data. In *SOSP*, pages 150–162, 1979.
20. Seth Gilbert, Nancy A. Lynch, and Alexander A. Shvartsman. Rambo: a robust, reconfigurable atomic memory service for dynamic networks. *Distributed Comput.*, 23(4):225–272, 2010.

21 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):123–149, 1991.

22 Maurice Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

23 Leander Jehl, Roman Vitenberg, and Hein Meling. SmartMerge: A New Approach to Reconfiguration for Atomic Storage. In *DISC*, pages 154–169, 2015.

24 Petr Kuznetsov, Thibault Rieutord, and Sara Tucci-Piergiovanni. Reconfigurable Lattice Agreement and Applications. *CoRR*, abs/1910.09264, 2019. `arXiv:1910.09264`.

25 Leslie Lamport. The Part-Time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

26 Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a state machine. *SIGACT News*, 41(1):63–73, 2010.

27 Matthieu Perrin. Concurrency and Consistency. In *Distributed Systems*. Elsevier, 2017. `doi:10.1016/B978-1-78548-226-7.50008-2`.

28 Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

29 Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-Free Replicated Data Types. In *SSS*, pages 386–400, 2011.

30 Jan Skrzypczak, Florian Schintke, and Thorsten Schütt. Linearizable State Machine Replication of State-Based CRDTs without Logs. *CoRR*, abs/1905.08733, 2019. `arXiv:1905.08733`.

31 Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi. Dynamic Reconfiguration: Abstraction and Optimal Asynchronous Solution. In *DISC*, pages 40:1–40:15, 2017.

Towards Distributed Two-Stage Stochastic Optimization

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

Noga Harlev
Technion - Israel Institute of Technology, Haifa, Israel
snogazur@campus.technion.ac.il

Taisuke Izumi
Nagoya Institute of Technology, Japan
t-izumi@nitech.ac.jp

Abstract

The *weighted vertex cover* problem is concerned with selecting a subset of the vertices that covers a target set of edges with the objective of minimizing the total cost of the selected vertices. We consider a variant of this classic combinatorial optimization problem where the target edge set is not fully known; rather, it is characterized by a probability distribution. Adhering to the model of *two-stage stochastic optimization*, the execution is divided into two stages so that in the first stage, the decision maker selects some of the vertices based on the probabilistic forecast of the target edge set. Then, in the second stage, the edges in the target set are revealed and in order to cover them, the decision maker can augment the vertex subset selected in the first stage with additional vertices. However, in the second stage, the vertex cost increases by some inflation factor, so the second stage selection becomes more expensive.

The current paper studies the two-stage stochastic vertex cover problem in the realm of *distributed graph algorithms*, where the decision making process (in both stages) is distributed among the vertices of the graph. By combining the stochastic optimization toolbox with recent advances in distributed algorithms for weighted vertex cover, we develop an algorithm that runs in time $O(\log(\Delta)/\varepsilon)$, sends $O(m)$ messages in total, and guarantees to approximate the optimal solution within a $(3+\varepsilon)$-ratio, where m is the number of edges in the graph, Δ is its maximum degree, and $0 < \varepsilon < 1$ is a performance parameter.

2012 ACM Subject Classification Theory of computation → Distributed computing models

Keywords and phrases weighted vertex cover, distributed graph algorithms, two-stage stochastic optimization, primal-dual

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.32

Funding *Yuval Emek*: This work has been funded in part by an Israeli Ministry of Science and Technology grant number 3-13565.
Taisuke Izumi: This work was supported by JST SICORP Grant Number JPMJSC1606 and JSPS KAKENHI Grant Number JP19K11824, Japan.

1 Introduction

Distributed computing models are adversarial in nature: if the future bears any kind of uncertainty, then we better prepare for the worst. While this approach is sensible in some situations, e.g., when dealing with faults in critical systems, it seems to be exaggerated in others: Should we still aim for the worst even if we hold reliable forecasts that point to more optimistic outcomes? Is it not a little bit paranoid to assume that the future is always determined by a malicious adversary?

© Yuval Emek, Noga Harlev, and Taisuke Izumi;
licensed under Creative Commons License CC-BY
23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller; Article No. 32; pp. 32:1–32:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

With this motivation in mind, the current paper adopts the *two-stage stochastic optimization* model, where the future uncertainty is resolved by nature's coin tosses rather than by a malicious adversary. This popular model is adjusted to the realm of *distributed graph algorithms* and applied to the *weighted vertex cover* problem. The adaptation of weighted vertex cover to the distributed stochastic setting makes it applicable to many real-life scenarios, especially in the context of network applications. For example, a distributed construction of a minimum vertex cover could come in handy for tasks such as monitoring traffic over high demand links. More often than not, the set of high demand links is not fully known in advance and decisions regarding the placement of monitors must be made based on probabilistic forecasts.

1.1 Distributed Two-Stage Stochastic Vertex Cover

The problem considered in this paper is a distributed version of the *two-stage stochastic vertex cover (2SVC)* problem. In its centralized version, the 2SVC problem is defined over an undirected graph $G = (V, E)$ and a vertex *cost* function $c : V \to \mathbb{R}_{\geq 0}$. Given a *target* edge subset $T \subseteq E$, the goal is to purchase a vertex subset $X \subseteq V$ that *covers* T in the sense that every edge in T has at least one endpoint in X.

The crux of the 2SVC problem is that the target T is not known explicitly; rather, it is drawn from a probability distribution π over 2^E. A 2SVC algorithm `Alg` constructs the covering vertex subset X in two *stages*: In the first stage, `Alg` constructs a vertex subset $X_1 \subseteq V$ based on π, but without knowing the realized target T. Then, in the second stage, T is revealed and `Alg` augments X_1 with a vertex subset $X_2 \subseteq V - X_1$ to obtain a valid cover $X = X_1 \cup X_2$ for T, but the vertices are now costlier: each vertex $v \in X_2$ costs $\sigma \cdot c(v)$, where $\sigma > 1$ is an *inflation factor* specified with the problem instance (together with G, c, and π). The objective is to minimize `Alg`'s total cost $c(X_1) + \sigma \cdot c(X_2)$ in expectation, where $c(V') = \sum_{v \in V'} c(v)$ for every $V' \subseteq V$ and the expectation is taken over the probability distribution π from which the target T is drawn and the random coin tosses of the algorithm (if any).

In the current paper, we adapt the 2SVC problem to the distributed setting where there is no centralized decision maker. Adhering to the standard assumptions in the domain of distributed graph algorithms (cf. [32]), the computation in both the first and second stages is distributed among the vertices in V that are identified with processing units that operate in synchronous *rounds* and can communicate with each other by exchanging messages of bounded size over the edges in E (referred to as the *CONGEST* model in [32]). This means that at the beginning of the first stage, the vertices hold no knowledge of the global topology of G. Moreover, we assume that π is a product distribution, so that $\pi(T) = (\prod_{e \in T} \pi(e)) \cdot (\prod_{e \in E - T}(1 - \pi(e)))$, and that it is provided to the vertices in a distributed manner: at the beginning of the first stage, vertex $v \in V$ is aware of the individual edge probability $\pi(e)$ if and only if e is incident on v.

The *approximation ratio* of a distributed 2SVC algorithm `Alg` is the minimum ρ such that for every graph $G = (V, E)$, cost function $c : V \to \mathbb{R}_{\geq 0}$, target (product) distribution π over 2^E, and inflation factor $\sigma > 1$, it is guaranteed that the expected cost paid by `Alg` is at most

$$\rho \times \min_{X_1 \subseteq V} \left(c(X_1) + \sum_{T \subseteq E} \pi(T) \cdot \min_{X_2 \subseteq V \,:\, X_1 \cup X_2 \text{ covers } T} \sigma \cdot c(X_2) \right) .$$

In other words, the performance of `Alg` is measured in comparison to an optimal (omnipotent) centralized benchmark.

1.2 Our Contribution

In this paper, we design a distributed algorithm, referred to as `Alg`, that for any $0 < \varepsilon < 1$, outputs a $(3 + \varepsilon)$-approximation for the 2SVC problem. Denoting the number of vertices in G by n and its maximum degree by Δ and assuming that the vertex costs $c(v)$ and performance parameter ε can be encoded using $O(\log n)$ bits, `Alg` completes the first stage in $O(\log(\Delta)/\varepsilon)$ rounds and the second stage in $O(1)$ rounds, sending a total of $O(1)$ messages per edge, each of size $O(\log n)$. We assume for simplicity that the vertices are equipped with unique IDs (encoded using $O(\log n)$ bits), but these can be easily replaced by any local symmetry breaking oracle (e.g., an arbitrary edge orientation). Beyond that, the vertices are not assumed to hold any global knowledge of the graph, including the parameters n and Δ.

Two-stage (and more generally, multi-stage) stochastic optimization is an important research domain (see Sec. 1.3) that, to the best of our knowledge, has not been studied yet in the context of distributed graph algorithms. We hope that the first step that the current paper makes into the interface between these fascinating research domains will ignite further exploration of distributed algorithms for stochastic graph theoretic problems.

1.3 Related Work

The field of stochastic optimization dates back to the mid-fifties with the seminal papers of Dantzig [12] and Beale [8] on stochastic linear programming and is studied extensively since then. Comprehensive accounts of stochastic programming under both continuous and discrete models can be found in [35, 27, 10].

Initiated in the paper of Dye, Stougie, and Tomasgard [14], the study of approximation algorithms for stochastic optimization problems has recently gained a lot of attention [21, 13, 34, 15, 19, 3]. Much of this literature deals with the *finite scenario* model, where the scenarios that may occur in the second stage are listed explicitly as part of the input. A different approach, sometimes called the *black-box* model, relies on sampling access to a probability distribution that is not necessarily provided explicitly. Shmoys and Swamy obtained approximation algorithms in this model for problems in the framework of two-stage [36] and multi-stage [38] stochastic optimization. They also wrote a broad survey on approximation algorithms for a large class of two-stage stochastic linear and integer programs in both the finite scenario and black-box models [39].

Among other methods, Shmoys and Swamy [38] used the *sample average approximation* approach that reduces many problems in the black-box model to their finite scenario counterpart. This approach essentially feeds the finite scenario algorithm with samples taken from the (implicit) probability distribution. We remark that although this approach has proven to be very successful for centralized algorithms [36, 38, 37], it seems to be less suitable for distributed settings.

As discussed later on, our work builds upon the (centralized) framework of Gupta et al. [18] for general stochastic optimization problems. This is a generic framework that given an approximation algorithm with certain desired properties for (the deterministic version) of a combinatorial optimization problem $\mathcal{P}$, yields an approximation algorithm for the two-stage (and multi-stage) stochastic versions of $\mathcal{P}$. Among other combinatorial optimization problems, Gupta et al. applied their framework to weighted vertex cover, obtaining a (centralized) 3-approximation 2SVC algorithm. This has been improved to a $(2 + \varepsilon)$-approximation by Srinivasan [37] using the sample average approximation approach.

In its deterministic version, vertex cover is a classic combinatorial optimization problem, listed among the 21 NP-hard problems in Karp's seminal paper [24]. The unweighted version of this problem can be approximated within a factor of 2 simply by finding a maximal

matching (see, e.g., [11]). A 2-approximation polynomial-time algorithm for the weighted version of vertex cover was introduced by Nemhauser and Trotter [30]. Bar-Yehuda and Even [5] presented the first linear-time 2-approximation algorithm using the primal-dual technique and later on [6], repeated this result to demonstrate their local ratio technique. The currently best known approximation for weighted vertex cover problem is $(2 - \Theta(1/\sqrt{\log n}))$ due to Karakostas [23]. Assuming the unique games conjecture, this cannot be improved much further as Khot and Regev [25] established a lower bound of $2 - \varepsilon$ for any constant $\varepsilon > 0$.

In the distributed setting, there is a long line of work on approximation algorithms for unweighted vertex cover and on the related maximal matching problem [22, 20, 31, 1, 33, 7, 16]. For the weighted version of vertex cover, Khuller et al. [26] present a $(2 + \varepsilon)$-approximation algorithm that runs in $O(\log \varepsilon^{-1} \log n)$ rounds. The algorithm of Kuhn et al. [29] obtains the same approximation ratio with an $O(\varepsilon^{-4} \log \Delta)$ run-time. Åstrand and Suomela [2] designed a 2-approximation algorithm whose run-time is $O(\Delta + \log^* W)$, where W is the maximum weight. A randomized 2-approximation algorithm that runs in $O(\log n + \log W)$ was presented by Grandoni et al. [17]. This has been improved to a run-time of $O(\log n)$ by Koufogiannakis and Young [28].

Building upon the classic local ratio technique, Bar-Yehuda et al. [4] have recently devised a deterministic $(2 + \varepsilon)$-approximation algorithm for weighted vertex cover with $O\left(\frac{\log \Delta}{\varepsilon \cdot \log \log \Delta}\right)$ run-time. This algorithm has been improved by Ben-Basat et el. [9] to obtain a run-time whose dependency on $1/\varepsilon$ is logarithmic rather than linear. With the right choice of the performance parameter ε, this leads to a deterministic 2-approximation algorithm for weighted vertex cover whose run-time is $O\left(\frac{\log n \cdot \log \Delta}{\log^2 \log \Delta}\right)$. As discussed further in the sequel, the approach behind the algorithms of [4, 9] inspires the main algorithmic building block of our 2SVC approximation algorithm.

1.4 Organization of the Paper

The remainder of this paper is organized as follows. Following some preliminary definitions presented in Sec. 2, we discuss in Sec. 3 the technical challenges that had to be overcome en route to developing our distributed 2SVC approximation algorithm. The algorithm is then developed in Sec. 4 and analyzed in Sec. 5. We conclude in Sec. 6 with some further discussion and open questions.

2 Preliminaries

Relaxed Vertex Cover. We now define a slightly different version of the 2SVC problem, called *relaxed 2SVC (r2SVC)*. In this version we are allowed to make partial payments for the vertices in each of the two stages based on the notion of a *payment* function $p : V \to \mathbb{R}_{\geq 0}$. We say that a payment function p *covers* the edge subset $F \subseteq E$ if the vertex subset $X^p = \{v \in V \mid p(v) \geq c(v)\}$ is a cover for F.

An algorithm $\texttt{Alg}^{\text{rel}}$ for the r2SVC problem constructs the payment functions $p_1 : V \to \mathbb{R}_{\geq 0}$ and $p_2 : V \to \mathbb{R}_{\geq 0}$ in the first and second stages, respectively, with the requirement that the function $p_1 + p_2$, defined so that $(p_1 + p_2)(v) = p_1(v) + p_2(v)$, covers the target edge subset T. The cost paid by $\texttt{Alg}^{\text{rel}}$ is $c(p_1) + \sigma \cdot c(p_2)$, where the cost $c(p)$ of a payment function $p : V \to \mathbb{R}_{\geq 0}$ is defined to be $c(p) = \sum_{v \in V} p(v)$.

By definition, any 2SVC algorithm can be transformed into a r2SVC algorithm with the same cost by setting $p_i(v) = c(v)$ if $v \in X_i$ and $p_i(v) = 0$ otherwise for $i = 1, 2$. Gupta et al. [18] established the converse direction: any r2SVC algorithm can be transformed into

a 2SVC algorithm with the same expected cost. Although their proof aims at centralized algorithms, it is straightforward to see that it holds also for the distributed version of the (r)2SVC problem, yielding the following lemma (a proof is added for completeness).

▶ **Lemma 2.1.** *Any distributed r2SVC algorithm* $\texttt{Alg}^{\text{rel}}$ *can be transformed into a distributed 2SVC algorithm* $\texttt{Alg}$ *with the same expected cost.*

Proof. Let p_1 and p_2 be the payment functions constructed by $\texttt{Alg}^{\text{rel}}$ in the first and second stages, respectively. We construct the 2SVC algorithm $\texttt{Alg}$ as follows. In the first stage $\texttt{Alg}$ includes vertex $v \in V$ in the vertex subset X_1 with probability $\min\{1, p_1(v)/c(v)\}$. In the second stage, $\texttt{Alg}$ includes v in the vertex subset X_2 if $v \in X^{p_1+p_2}$ and it was not already selected in the first stage. By the linearity of expectation, the expected payment made by $\texttt{Alg}$ in each stage is up-bounded by the payment made by $\texttt{Alg}^{\text{rel}}$ in the same stage. ◀

Following Lem. 2.1, we focus hereafter on designing a distributed algorithm for the r2SVC problem.

The Framework of [18]. Our distributed algorithm for the r2SVC problem is based on the (centralized) *boosted-sampling* algorithm of Gupta et al. [18] for that problem.[1] This is a randomized algorithm that is compiled from two algorithmic building blocks. The first building block is a *covering payment* algorithm, referred to as $\texttt{Alg}^{\text{pay}}$, that given an edge subset $F \subseteq E$, constructs a payment function $p : V \to \mathbb{R}_{\geq 0}$ that covers F.[2] We say that $\texttt{Alg}^{\text{pay}}$ is an α-*approximation* covering payment algorithm if the cost of the payment function p constructed by $\texttt{Alg}^{\text{pay}}$ is guaranteed to satisfy $c(p) \leq \alpha \cdot c(\texttt{Opt}(F))$, where $\texttt{Opt}(\text{F})$ is an optimal cover for F.

The second building block of the boosted-sampling algorithm is an *augmentation algorithm*, referred to as $\texttt{Alg}^{\text{aug}}$, that given two edge subsets $F, F' \subseteq E$ and a payment function $p : V \to \mathbb{R}_{\geq 0}$ that covers F, constructs a payment function p' so that $p + p'$ covers F'.

In the first stage, the boosted-sampling algorithm creates a random edge subset S by sampling each edge $e \in E$ with probability $\min\{1, \sigma\pi(e)\}$. Following that, it invokes $\texttt{Alg}^{\text{pay}}$ on S to obtain a payment function p_1 that covers S. In the second stage, when the actual edge subset T is revealed, the boosted-sampling algorithm invokes $\texttt{Alg}^{\text{aug}}$ on (S, T, p_1) to construct the payment function p_2 so that $p_1 + p_2$ covers T.

An essential ingredient in the analysis of the boosted-sampling algorithm is the notion of *cost-sharing* functions. These functions provide a useful way to bound the overall cost by allocating it to the edges that need to be covered. Formally, a function $\xi : 2^E \times E \to \mathbb{R}_{\geq 0}$ is said to be a *cost-sharing* function if it satisfies the following two properties for any edge subset $F \subseteq E$:

P1. $\xi(F, e) = 0$ for any edge $e \in E - F$; and

P2. $\sum_{e \in F} \xi(F, e) \leq c(\texttt{Opt}(F))$.

The cost-sharing function ξ is said to be β-*unistrict* with respect to $\texttt{Alg}^{\text{pay}}$ and $\texttt{Alg}^{\text{aug}}$ if it satisfies the following property for any edge subset $F \subset E$ and edge $e \in E - F$:

P3. $\beta \cdot \xi(F \cup \{e\}, e) \geq c(\texttt{Alg}^{\text{aug}}(F, \{e\}, \texttt{Alg}^{\text{pay}}(F)))$.

[1] As mentioned in Sec. 1.3, Gupta et al. develop a generic framework and the boosted-sampling algorithm is actually suitable for many different combinatorial optimization problems. Since our focus in the current paper is restricted to weighted vertex cover, we present an adaptation of this algorithm to that specific problem.

[2] We assume that the underlying graph $G = (V, E)$ and the vertex cost function c are fixed and do not explicitly pass them as arguments to the various algorithms described hereafter.

$$\text{(P)} \quad \min \sum_{v \in V} x_v c(v) \qquad \text{s.t.} \quad x_u + x_v \geq 1 \quad \forall (u,v) \in E \qquad x_u \geq 0 \quad \forall v \in V$$

$$\text{(D)} \quad \max \sum_{e \in E} y_e \qquad \text{s.t.} \quad \sum_{e: v \in e} y_e \leq c(v) \quad \forall v \in V \qquad y_e \geq 0, \quad \forall e \in E$$

Figure 1 The linear program relaxation of weighted vertex cover (P) and its dual program (D), which is actually the linear program relaxation of the *b-matching* problem.

In other words, the cost of augmenting the payment function p constructed by $\texttt{Alg}^{\text{pay}}$ for covering F to a payment function $p + p'$ that covers e as well, is at most $\beta \cdot \xi(F \cup \{e\}, e)$.

▶ **Theorem 2.2** ([18]). *Given an α-approximation covering payment algorithm $\texttt{Alg}^{\text{pay}}$, an augmentation algorithm $\texttt{Alg}^{\text{aug}}$ and a cost-sharing function that is β-unistrict with respect to $\texttt{Alg}^{\text{pay}}$ and $\texttt{Alg}^{\text{aug}}$, the boosted-sampling algorithm provides an $(\alpha + \beta)$-approximation algorithm for the r2SVC problem.*

Additional Notation. Throughout, we denote the number of vertices and the number of edges in the underlying graph $G = (V, E)$ by $n = |V|$ and $m = |E|$, respectively. Let $\deg(v)$ be the degree of vertex v in G and let $\Delta = \max_{v \in V} \deg(v)$ be the maximum degree.

3 Technical Challenges

In their centralized 2SVC algorithm, Gupta et al. [18] construct the first-stage solution using the following classic (centralized) primal-dual algorithm for weighted vertex cover (refer to Fig. 1 for the weighted vertex cover linear program relaxation and its dual program): The algorithm continuously raises the dual variables associated with the edges, concurrently for all of them, until a dual constraint associated with some vertex v becomes tight; at this point, v joins the vertex cover and the edges incident on v are *frozen* so that their dual variables will not be raised any further. The algorithm terminates when all edges are frozen.

A key ingredient in the analysis of Gupta et al. is that the dual variables can serve as a 1-unistrict cost-sharing function. This powerful observation utilizes the *identical runs* property stating that for any $F \subset E$ and $e \in E - F$, the runs of the algorithm on F and on $F \cup \{e\}$ are identical (with respect to the dual variables of the edges in F) up to the point when the dual variable of e becomes frozen.

The recent weighted vertex cover algorithm of Bar-Yehuda et al. [4] (see also the algorithm of Ben-Basat et al. [9]) can be viewed as a distributed implementation of this primal-dual approach.[3] They cleverly replace the centralized method of continuously (and concurrently) increasing the dual variables by a stepwise increase based on a *request-response* iterative process. An inherent property of the algorithms of [4, 9] is that in each step of the iterative process, the amount requested by vertex v from an adjacent vertex u, and hence also the amount responded by u to v's request, depend on the number of edges incident on v that have not been frozen yet. Consequently, the runs of the algorithm on $F \subset E$ and on $F \cup \{e\}$,

[3] The authors of [4, 9] present their algorithms using the equivalent local ratio approach.

for some $e \in E - F$, may differ from each other already at the early stages of the execution. As the amounts of responses sent over edge $e = (u, v)$ determine the value of the dual variable associated with e, this distributed algorithm does not satisfy the aforementioned identical runs property.

To overcome this obstacle, we develop a variant of [4]'s algorithm that does satisfy the identical runs property. This is done by grouping the iterative process' steps into *phases* so that vertex v requests the same amount throughout all steps of the phase. The crucial point here is that this amount is now fixed in advance and in particular, does not depend on the number of unfrozen edges incident on v. At the same time, our algorithm preserves the primal-dual structure in a way that allows us to efficiently combine it with an augmentation (second stage) algorithm. The main challenge is then to show that the dual variables serve as a $(1 + \varepsilon)$-unistrict cost-sharing function. We prove that our algorithm is only $O(\log \log \Delta)$ factor slower than the original algorithm of [4] (whose run-time is optimal as a function of Δ). Moreover, our algorithm has a message complexity of $O(m)$, an improvement over the algorithm of [4] that sends $O\left(\frac{m \log \Delta}{\varepsilon \cdot \log \log \Delta}\right)$ messages.

4 A Distributed Algorithm for 2SVC

In this section, we present our distributed 2SVC algorithm `Alg`. As mentioned in Sec. 2, we actually present an algorithm for the r2SVC problem (Lem. 2.1 ensures that it can be transformed into a 2SVC algorithm) that can be viewed as a distributed implementation of the boosted-sampling algorithm of [18].

Similarly to the centralized version of the boosted-sampling algorithm, our distributed algorithm also constructs the random edge subset S by sampling each edge $e \in E$ with probability $\min\{1, \sigma\pi(e)\}$, only that now this sampling is done in a decentralized manner by one of e's endpoints (say, the one with the smaller ID). Following this sampling process, the vertices know which of their incident edges are included in S.

The first stage is then completed by running the distributed covering payment algorithm $\texttt{Alg}^{\text{pay}}$ presented in Sec. 4.1 on S, generating a payment function p_1 that covers S. In the second stage when the target edge subset $T \subseteq E$ is revealed, the vertices run the distributed augmentation algorithm $\texttt{Alg}^{\text{aug}}$ presented in Sec. 4.2 on S, T, and p_1 to obtain a payment function p_2 so that $p_1 + p_2$ covers T. For clarity of the exposition, we first implement our algorithms so that they may send messages over each edge in every round without taking the message size into account. Then, in Sec. 4.3, we explain how $\texttt{Alg}^{\text{pay}}$ and $\texttt{Alg}^{\text{aug}}$ can be modified to send $O(m)$ messages in total, each of size $O(\log n)$.

4.1 A Distributed Covering Payment Algorithm

Our distributed covering payment algorithm $\texttt{Alg}^{\text{pay}}$ gets as input an edge set F and constructs a payment function p that covers F. Fix some sufficiently small performance parameter $\varepsilon > 0$ (the relation of the performance parameter ε fixed here to the guaranteed approximation ratio is revealed later on). $\texttt{Alg}^{\text{pay}}$ works in phases, each consisting of $3 \cdot \lceil 2/\varepsilon \rceil$ rounds that are divided into $\lceil 2/\varepsilon \rceil$ contiguous round triples referred to as *steps*. Each step begins with a *request* round followed by a *response* round, in which the vertices exchange messages referred to as *requests* and *responses*, respectively. The last (third) round in the step is a *status* round in which the vertices report whether they remain *active* (initially, all vertices are active). Every active vertex $v \in V$ maintains a *weight* variable $w(v)$ which is initialized with $c(v)$ and is monotonically non-increasing throughout the steps of the algorithm. In addition, vertex v maintains a set $A(v)$ of active neighbors.

Consider some active vertex $v \in V$. We use (j, i) to denote step $i = 0, 1, \ldots, \lceil 2/\varepsilon \rceil - 1$ of phase j. In the request round of step (j, i), vertex v sends to each active neighbor $u \in A(v)$ the message

$$\texttt{request}_{j,i}(v, u) \;=\; 2^j \cdot \frac{\varepsilon c(v)}{\deg(v)} \,,$$

where recall that $\deg(v)$ denotes the degree of v in the underlying graph G. In the following response round, v processes the request messages received from its active neighbors one-by-one (in an arbitrary order). For each request message $\texttt{request}_{j,i}(u, v)$, v sends a response message

$$\texttt{response}_{j,i}(v, u) \;=\; \min\left\{\texttt{request}_{j,i}(u, v), w(v) - \varepsilon c(v)\right\}$$

and subtracts $\texttt{response}_{j,i}(v, u)$ from $w(v)$. We say that the request $\texttt{request}_{j,i}(u, v)$ is *fully responded* if $\texttt{response}_{j,i}(v, u) = \texttt{request}_{j,i}(u, v)$. Notice that this update rule ensures that $w(v) \geq \varepsilon c(v)$ throughout the response round.

In the status round, v updates the weight variable $w(v)$ by subtracting $\texttt{response}_{j,i}(u, v)$ from $w(v)$ for each response message received from an active neighbor $u \in A(v)$. Upon completion of this update process, if $w(v) \leq \varepsilon c(v)$, then v becomes inactive, sends a designated `inactive` message to its active neighbors, and marks itself as a *covering* vertex (the role of the covering vertices is revealed soon). A vertex whose active neighbor set is empty also becomes inactive. We use $w_{inact}(v)$ to denote the remaining weight of v when it becomes inactive.

The payment function $p : V \to \mathbb{R}_{\geq 0}$ constructed by $\texttt{Alg}^{\text{pay}}$ is defined by setting

$$p(v) \;=\; \min\left\{\frac{c(v) - w_{inact}(v)}{1 - \varepsilon}, c(v)\right\} \,. \tag{1}$$

This means that $p(v)$ is always up-bounded by $c(v)$ and that $p(v) = c(v)$ if and only if v ends up as a covering vertex. Refer to Pseudocode 1 for a pseudocode description of $\texttt{Alg}^{\text{pay}}$.

4.2 A Distributed Augmentation Algorithm

Our distributed augmentation algorithm $\texttt{Alg}^{\text{aug}}$ gets as input two edge subsets $F, F' \subseteq E$ and a payment function $p : V \to \mathbb{R}_{\geq 0}$ that covers F, and constructs a payment function $p' : V \to \mathbb{R}_{\geq 0}$ so that $p + p'$ covers F' as follows. Define the *reduced* cost of each vertex $v \in V$ to be $c(v) - p(v)$. For each edge $(v_a, v_b) \in F'$, $\texttt{Alg}^{\text{aug}}$ *selects* the endpoint v_i, $i \in \{a, b\}$, with the smaller reduced cost $c(v_i) - p(v_i)$, breaking ties arbitrarily (say, by the vertex IDs). The payment function p' is then defined by setting

$$p'(v) = \begin{cases} c(v) - p(v), & \text{if } v \text{ is selected by } \texttt{Alg}^{\text{aug}} \\ 0, & \text{otherwise} \end{cases} \,.$$

This means that

$$c\left(\texttt{Alg}^{\text{aug}}\left(F, F', p\right)\right) \;\leq\; \sum_{(v_a, v_b) \in F'} \min\left\{c(v_a) - p(v_a), c(v_b) - p(v_b)\right\} \,. \tag{2}$$

Algorithm 1 A covering payment algorithm, code for vertex v.

```
for j = 0, 1, ... do
    for i = 0 to ⌈2/ε⌉ − 1 do
        Request Round
        for each u ∈ A(v) do
            request_{j,i}(v, u) ← 2^j · εc(v)/deg(v)
            Send request_{j,i}(v, u) to u
        Response Round
        for each u ∈ A(v) do
            response_{j,i}(v, u) ← min {request_{j,i}(u, v), w(v) − εc(v)}
            w(v) ← w(v) − response_{j,i}(v, u)
            Send response_{j,i}(v, u) to u
        Status Round
        for each u ∈ A(v) do
            w(v) ← w(v) − response_{j,i}(u, v)
        if w(v) ≤ εc(v) then
            Send inactive to all neighbors
            return p(v) ← c(v)                    ▷ v is marked as a covering vertex
        for each inactive message received from u ∈ A(v) do
            A(v) ← A(v) − {u}
        if A(v) = ∅ then
            return p(v) ← 1/(1−ε) (c(v) − w(v))
```

4.3 Fewer and Smaller Messages

While $\texttt{Alg}^{\text{aug}}$ requires a single round of communication and hence, does not send more than $O(m)$ messages in total, the aforementioned implementation of $\texttt{Alg}^{\text{pay}}$ dictates sending messages over each edge in every step which sums up to $\Omega(m \log(\Delta)/\varepsilon)$ messages. Here we show that $\texttt{Alg}^{\text{pay}}$ can be modified to send $O(1)$ messages over each edge throughout the execution, which accounts for a total of $O(m)$ messages as well. We then explain how the message size (of both $\texttt{Alg}^{\text{pay}}$ and $\texttt{Alg}^{\text{aug}}$) can be bounded by $O(\log n)$.

The modified $\texttt{Alg}^{\text{pay}}$ starts with a *handshake* round in which each vertex $v \in V$ sends its cost $c(v)$ and degree $\deg(v)$ to all its neighbors. The next messages that v is certain to send are the `inactive` messages sent when v becomes inactive. Consider some neighbor u of v and suppose that it still did not receive an `inactive` message from v. Using the handshake information, vertex u can compute the value of each request message $\texttt{request}_{j,i}(v,u) = 2^j \cdot \varepsilon c(v)/\deg(v)$ without actually receiving it from v. Moreover, as long as u does not hear otherwise from v, it infers that v fully responds to its own requests, namely, that $\texttt{response}_{j,i}(v,u) = \texttt{request}_{j,i}(u,v)$, again, without actually receiving v's response. If v cannot fully respond to u's request in some step (j,i), then v sends the partial response $\texttt{response}_{j,i}(v,u)$ as in the original implementation of $\texttt{Alg}^{\text{pay}}$, but notice that this will happen at most once as v must become inactive in step (j,i).

With this modification, a vertex $v \in V$ sends messages over an incident edge (v,u) in three occasions: (1) a message encoding $c(v)$ and $\deg(v)$ during the designated handshake round of $\texttt{Alg}^{\text{pay}}$; (2) a message encoding a partial response to $\texttt{request}_{j,i}(u,v)$ during the (single) step (j,i) in which v becomes inactive; and (3) a message encoding the reduced cost

of v during the execution of $\texttt{Alg}^{\text{aug}}$ in the second stage. The message sent in occasion (1) is of size $O(\log n)$ by the assumption on the vertex costs. To ensure that the messages sent during occasions (2) and (3) are also of size $O(\log n)$, we round down their numeric content to the next power of $1+\varepsilon$. This affects the approximation ratio analyzed in Sec. 5 by a factor of at most $1+\varepsilon$.

5 Analysis

In this section, we analyze the distributed implementation $\texttt{Alg}$ of the boosted-sampling algorithm presented in Sec. 4, establishing the following theorem.

▶ **Theorem 5.1.** *For any sufficiently small $\varepsilon > 0$, $\texttt{Alg}$ completes the first stage in $O(\log(\Delta)/\varepsilon)$ rounds and the second stage in $O(1)$ rounds and guarantees to return a $(3+O(\varepsilon))$-approximation for the r2SVC problem.*

To obtain an approximation ratio of $(3+\varepsilon')$ for some $0 < \varepsilon' < 1$ (as promised in Sec. 1), one merely has to set the performance parameter ε used by $\texttt{Alg}$ so that $\varepsilon \leftarrow \varepsilon'/\kappa$ for the appropriate constant κ hidden in the O-notation in the statement of Thm. 5.1. We start the analysis by proving that $\texttt{Alg}^{\text{pay}}$ constructs a covering payment function.

▶ **Lemma 5.2.** *When invoked on an edge subset $F \subseteq E$, $\texttt{Alg}^{\text{pay}}$ constructs a payment function p that covers F.*

Proof. When a vertex $v \in V$ becomes inactive, either (1) its active neighbor set is empty; or (2) its weight variable $w(v)$ satisfies $w(v) \leq \varepsilon c(v)$. In the latter case, v is a covering vertex with $p(v) = c(v)$. The assertion follows by the design of $\texttt{Alg}^{\text{pay}}$ ensuring that if $(u,v) \in F$, then it cannot be the case that both u and v become inactive due to the former reason. ◀

Let $w_j(v)$ and $A_j(v)$ denote the weight and active neighbor set of v at the beginning of phase j, respectively, and let $\deg_j^A(v)$ denote the size of $A_j(v)$. The following two lemmas play a crucial role in proving that $\texttt{Alg}^{\text{pay}}$ is a $(2+O(\varepsilon))$-approximation covering payment algorithm.

▶ **Lemma 5.3.** *For every phase j in the run of $\texttt{Alg}^{\text{pay}}$ and for every vertex $v \in V$ that is active at the beginning of phase j, it holds that $\deg_j^A(v) \leq \deg(v)/2^j$.*

Proof. Consider some vertex $v \in V$ and recall that $\deg_0^A(v)$ is the degree of v in (V,F). Since $\deg_0^A(v)$ is up-bounded by $\deg(v)$, the assertion holds for phase $j=0$. Assume by contradiction that $\deg_j^A(v) > \deg(v)/2^j$ for some phase $j > 0$ where v is still active. This means that v had more than $\deg(v)/2^j$ active neighbors throughout phase $j-1$ and in each of the $\lceil 2/\varepsilon \rceil$ steps of that phase, more than $\deg(v)/2^j$ requests of v were fully responded. Since $\texttt{request}_{j-1,i}(v,u) = 2^{j-1} \cdot \varepsilon c(v)/\deg(v)$ for every $u \in A_{j-1}(v)$ and $i = 0,1,\ldots,\lceil 2/\varepsilon \rceil - 1$, it follows that

$$w_j(v) \;<\; w_{j-1}(v) - \lceil 2/\varepsilon \rceil \cdot \frac{\deg(v)}{2^j} \cdot 2^{j-1} \cdot \frac{\varepsilon c(v)}{\deg(v)} \;\leq\; w_{j-1}(v) - c(v) \;\leq\; 0\,.$$

Ergo, v must become inactive by the beginning of phase j, in contradiction to the assumption. ◀

▶ **Lemma 5.4.** *For every vertex $v \in V$, the weight variable $w(v)$ remains non-negative throughout the run of $\texttt{Alg}^{\text{pay}}$.*

Proof. Since the weight of v does not decrease once v becomes inactive, it suffices to show that $w(v)$ remains non-negative as long as v is active. In each step the weight of v can decrease only during the response round and the status round. Recall that $w(v) \geq \varepsilon c(v)$ throughout every response round, therefore we are left to show that in every status round, the amount decreased from $w(v)$ is not larger than $\varepsilon c(v)$.

Consider some step (j, i). Each response message $\texttt{response}_{j,i}(u, v)$ that v processes during this step is not larger than $\texttt{request}_{j,i}(v, u) = 2^j \cdot \varepsilon c(v)/\deg(v)$. Therefore, it remains to show that v receives at most $\deg(v)/2^j$ response messages in this step. As the number of active neighbors is monotonically non-increasing, this follows from Lem. 5.3 ensuring that v has at most $\deg(v)/2^j$ active neighbors at the beginning of phase j. ◀

For the purpose of further analysis, we define the functions $y_{j,i} : 2^E \times E \to \mathbb{R}_{\geq 0}$ and $y : 2^E \times E \to \mathbb{R}_{\geq 0}$ as follows. Consider a run of $\texttt{Alg}^{\text{pay}}$ on an edge subset $F \subseteq E$. For each step (j, i) in this run and edge $e = (u, v) \in E$, let $y_{j,i}(F, e) = \texttt{response}_{j,i}(u, v) + \texttt{response}_{j,i}(v, u)$ and let

$$y(F, e) \;=\; \sum_{j=0,1,\ldots,\, i=0,1,\ldots,\lceil 2/\varepsilon \rceil - 1} y_{j,i}(F, e)\,.$$

We generalize the last definition from single edges e to edge subsets $J \subseteq E$, defining $y(F, J) = \sum_{e \in J} y(J, e)$. In addition, let $F(v) = \{e \in F : v \in e\}$ denote the set of edges in F incident on vertex $v \in V$. Notice that for every vertex $v \in V$, the sum of response messages sent over edges in $F(v)$ equals the total amount decreased from $w(v)$ throughout the execution, which yields the following observation.

▶ **Observation 5.5.** *For every vertex $v \in V$, the function y satisfies $y(F, F(v)) = c(v) - w_{inact}(v)$.*

This facilitates approximating $p(v)$ by $y(F, F(v))$.

▶ **Lemma 5.6.** *When invoked on an edge subset $F \subseteq E$, $\texttt{Alg}^{\text{pay}}$ constructs a payment function p that satisfies*

$$(1 - \varepsilon) \cdot p(v) \;\leq\; y(F, F(v)) \;\leq\; p(v)$$

for every vertex $v \in V$.

Proof. Recall that $p(v) = \min\left\{\frac{1}{1-\varepsilon}(c(v) - w_{inact}(v)), c(v)\right\}$ (see (1)). If $\frac{1}{1-\varepsilon}(c(v) - w_{inact}(v)) \leq c(v)$, then the assertion holds by Obs. 5.5 implying that $p(v) = \frac{1}{1-\varepsilon} \cdot y(F, F(v))$. So, consider the case where $\frac{1}{1-\varepsilon}(c(v) - w_{inact}(v)) > c(v)$ which means that $w_{inact}(v) < \varepsilon \cdot c(v)$. Lem. 5.4 ensures that $w_{inact}(v) \geq 0$, thus by combining it with Obs. 5.5 we conclude that $(1 - \varepsilon) \cdot c(v) < y(F, F(v)) \leq c(v)$. The assertion follows as $p(v) = c(v)$ in this case. ◀

Employing Obs. 5.5 we can establish also the following lemma.

▶ **Lemma 5.7.** *The function $y : 2^E \times E \to \mathbb{R}_{\geq 0}$ satisfies $y(F, F) \leq c(\texttt{Opt}(F))$.*

Proof. For every vertex $v \in V$, Lem. 5.4 guarantees that $w_{inact}(v) \geq 0$, hence, in conjunction with Obs. 5.5, we get that $y(F, F(v)) \leq c(v)$. It follows then that the variables $y(F, e)$, $e \in F$, constitute a feasible solution for the dual program of the linear program relaxation of weighted vertex cover on graph (V, F) with cost function $c(\cdot)$ (refer to Fig. 1 for that program). The assertion follows as a consequence of weak duality. ◀

We are now ready to establish the approximation ratio of $\texttt{Alg}^{\text{pay}}$.

▶ **Lemma 5.8.** $\texttt{Alg}^{\text{pay}}$ *is a* $(2+O(\varepsilon))$*-approximation covering payment algorithm.*

Proof. The payment function p constructed by $\texttt{Alg}^{\text{pay}}$ satisfies

$$\sum_{v\in V} p(v) \;\le\; (1+O(\varepsilon))\sum_{v\in V} y(F,F(v)) \;=\; 2\cdot(1+O(\varepsilon))\cdot y(F,F) \;\le\; (2+O(\varepsilon))\cdot c(\texttt{Opt}(F))\,,$$

where the first transition is due to Lem. 5.6 and the last transition follows from Lem. 5.7. ◀

It follows directly from the design of $\texttt{Alg}^{\text{aug}}$ that given two edge subsets $F, F' \subseteq E$ and a payment function $p : V \to \mathbb{R}_{\ge 0}$ that covers F, the payment function p' constructed by $\texttt{Alg}^{\text{aug}}$ when invoked on (F, F', p) augments p so that $p + p'$ covers F'. So, $\texttt{Alg}^{\text{aug}}$ is a valid augmentation algorithm (that runs in $O(1)$ rounds). The desired bound on the approximation ratio of $\texttt{Alg}$ can now be established by identifying a proper cost-sharing function.

▶ **Lemma 5.9.** *The function* $y : 2^E \times E \to \mathbb{R}_{\ge 0}$ *is a* $(1+O(\varepsilon))$*-unistrict cost-sharing function with respect to* $\texttt{Alg}^{\text{pay}}$ *and* $\texttt{Alg}^{\text{aug}}$*.*

Proof. We need to show that y satisfies properties P1–P3 (see Sec. 2) with respect to $\texttt{Alg}^{\text{pay}}$ and $\texttt{Alg}^{\text{aug}}$. When $\texttt{Alg}^{\text{pay}}$ is invoked on an edge subset $F \subseteq E$, no messages are sent over edges that are not in F, so property P1 clearly holds. Moreover, Lem. 5.7 ensures that $\sum_{e\in F} y(F,e) \le c(\texttt{Opt}(F))$, thus property P2 holds as well.

For property P3, we fix some $F \subset E$ and $e \in E - F$ and show that

$$c\left(\texttt{Alg}^{\text{aug}}(F,\{e\},\texttt{Alg}^{\text{pay}}(F))\right) \le (1+O(\varepsilon))\cdot y(F\cup\{e\},e)\,.$$

To that end, we define the following additional notation: Given an edge subset $J \subseteq F$ and a step (j,i), let

$$y_{<(j,i)}(F,J) \;=\; \sum_{(j',i')<(j,i)}\;\sum_{e\in J} y_{j',i'}(F,e)\,,$$

where we use the relation $<$ to denote the lexicographic order, and let

$$y_{\le(j,i)}(F,J) \;=\; y_{<(j,i)}(F,J) + \sum_{e\in J} y_{j,i}(F,e)\,.$$

Consider two runs of $\texttt{Alg}^{\text{pay}}$: run R^1 on F and run R^2 on $F\cup\{e\}$. Let (j,i) be the first step in which an endpoint of e becomes inactive during R^2. Let $R^1_{j,i}$ and $R^2_{j,i}$ denote the runs R^1 and R^2 up to the beginning of step (j,i), respectively.

Observe that $R^1_{j,i}$ and $R^2_{j,i}$ are identical in the sense that $y_{<(j,i)}(F,J) = y_{<(j,i)}(F\cup\{e\},J)$ for any $J \subseteq F$. In other words, the same response messages are sent over every edge $e' \in F$ throughout $R^1_{j,i}$ and $R^2_{j,i}$. This is due to the fact that the request messages that (an active) vertex $v \in V$ sends in every step (j',i') do not depend on the input of $\texttt{Alg}^{\text{pay}}$, but only on $\deg(v)$ and j'. Since both endpoints of edge e' are active throughout $R^2_{j,i}$, it follows that the request messages sent over every edge in F are the same in $R^1_{j,i}$ and $R^2_{j,i}$, hence so are the response messages.

Let u be the endpoint of e that becomes inactive in step (j,i) of run R^2 (if both endpoints of e become inactive in this step, then take u to be one of them) and recall that u is marked as a covering vertex. We argue that for each edge $e' \in F(u)$, it holds that $y_{j,i}(F\cup\{e\},e') \le y_{j,i}(F,e')$. This follows from the fact that in step (j,i) of run R^2, once $w(u)$ reaches the $\varepsilon\cdot c(u)$ threshold, it starts sending response messages with 0 amount over the edges in $F(u)$, while in step (j,i) of R^1, u may still send response messages with positive amounts over these edges. Therefore, $y_{\le(j,i)}(F\cup\{e\},F(u)) \le y_{\le(j,i)}(F,F(u))$.

At the end of step (j,i) of R^2 we know that u is inactive and $w(u) \leq \varepsilon c(u)$, so the total amount decreased from $w(u)$ throughout this run is at least $(1-\varepsilon)\cdot c(u)$. Therefore, the amount decreased from $w(u)$ throughout R^2 due to messages sent over edges other than e is at least $(1-\varepsilon)\cdot c(u) - y(F \cup \{e\}, e)$, hence

$$y_{\leq(j,i)}(F, F(u)) \;\geq\; y_{\leq(j,i)}(F \cup \{e\}, F(u)) \;\geq\; (1-\varepsilon)\cdot c(u) - y(F \cup \{e\}, e)\,.$$

Consequently, the total amount decreased from $w(u)$ throughout R^1 is at least $(1-\varepsilon)\cdot c(u) - y(F \cup \{e\}, e)$, thus, at the end of R^1

$$w_{inact}(u) \;\leq\; c(u) - ((1-\varepsilon)\cdot c(u) - y(F \cup \{e\}, e)) \;=\; \varepsilon c(u) + y(F \cup \{e\}, e)\,. \tag{3}$$

Let p be the payment function constructed by $\texttt{Alg}^{\text{pay}}(F)$ and let p' be the payment function constructed by $\texttt{Alg}^{\text{aug}}(F, \{e = (u,v)\}, p)$. As p is a covering payment function for F, the bound in (2) implies that $c(p') \leq \min\{c(u) - p(u), c(v) - p(v)\} \leq c(u) - p(u)$. Recalling the definition of p (1), if $p(u) = c(u)$, then $c(p') = 0$ and the assertion trivially holds. Otherwise, $p(u) = \frac{c(u) - w_{inact}(u)}{1-\varepsilon}$ which means that

$$c(p') \;\leq\; c(u) - p(u) \;=\; c(u) - \frac{c(u) - w_{inact}(u)}{1-\varepsilon} \;=\; \frac{w_{inact}(u) - \varepsilon c(u)}{1-\varepsilon}\,.$$

Combined with (3), we conclude that

$$c(p') \;\leq\; \frac{y(F \cup \{e\}, e)}{1-\varepsilon} \;=\; (1 + O(\varepsilon)) \cdot y(F \cup \{e\}, e)\,,$$

thus establishing the assertion. ◀

Combining Lem. 5.8 and 5.9 with Thm. 2.2, we conclude that `Alg` returns a $(3 + O(\varepsilon))$-approximation for the r2SVC problem, as promised in Thm. 5.1. It remains to analyze `Alg`'s run-time.

▶ **Lemma 5.10.** `Alg` *completes the first stage in* $O(\log(\Delta)/\varepsilon)$ *rounds.*

Proof. The edge sampling performed at the beginning of the first stage takes $O(1)$ rounds, so it is left to bound the run-time of $\texttt{Alg}^{\text{pay}}$. For every vertex $v \in V$, Lem. 5.3 ensures that after $O(\log \deg_0^A(v)) = O(\log \deg(v))$ phases, the neighbor set of v must be empty which means that v must become inactive. Since each phase consists of $3 \cdot \lceil 2/\varepsilon \rceil$ rounds, vertex v completes the execution in $O(\log(\deg(v))/\varepsilon) = O(\log(\Delta)/\varepsilon)$ rounds. ◀

The second stage includes only the execution of $\texttt{Alg}^{\text{aug}}$, that requires a single round of communication. By that we complete the run-time analysis of `Alg`, thus establishing Thm. 5.1.

6 Discussion and Open Questions

In this paper, we restrict our attention to the case where the target edge set T is drawn from a product distribution, which means that the edge sampling events are independent. The more general case, that allows the edges to exhibit complicated dependencies, is studied extensively in the centralized two-stage stochastic optimization literature and it will be interesting to come up with a distributed method that supports it. This requires overcoming the following obstacle though: it is not clear how to sample the target edge set T in a distributed fashion if the events $e \in T$ and $e' \in T$ depend on each other for edges e and e' that are far away in the graph.

A related open question is concerned with developing a meaningful notion of a finite scenario model for distributed two-stage stochastic optimization. Here too it is not clear how the vertices can select a target edge set in a correlated manner. Overcoming this obstacle may also be the first step towards a distributed sample average approximation approach.

It would be interesting to extend our results to the *k-stage* stochastic optimization model as well. In this model, the vertex cover is constructed in k (that may be larger than 2) stages so that in each stage, the probability distribution gets refined, but the inflation factor is increased. Another interesting extension would be a model where the revelation of the target edge set is not necessarily done for all vertices at once, but rather, different vertices may receive their local image of the target edge set's realization in different rounds.

Besides that, we would like to examine whether the run-time complexity of our algorithm can be improved. Specifically, reducing the dependency on ε from linear to logarithmic would yield a distributed 3-approximation algorithm for the 2SVC problem whose run-time is polylogarithmic (cf. [9]), although we have yet to find a way to do so. Finally, we believe that there are great opportunities in adapting other graph theoretic optimization problems to the distributed stochastic setting.

References

1. Matti Åstrand, Patrik Floréen, Valentin Polishchuk, Joel Rybicki, Jukka Suomela, and Jara Uitto. A local 2-approximation algorithm for the vertex cover problem. In *International Symposium on Distributed Computing*, pages 191–205. Springer, 2009.
2. Matti Åstrand and Jukka Suomela. Fast distributed approximation algorithms for vertex cover and set cover in anonymous networks. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 294–302. ACM, 2010.
3. Ilke Bakir, Natashia Boland, Brian Dandurand, and Alan Erera. Scenario set partition dual bounds for multistage stochastic programming: A hierarchy of bounds and a partition sampling approach. *Optimization Online*, 2016.
4. Reuven Bar-Yehuda, Keren Censor-Hillel, and Gregory Schwartzman. A Distributed (2+ ε)-Approximation for Vertex Cover in O (log Δ/ε log log Δ) Rounds. *Journal of the ACM (JACM)*, 64(3):23, 2017.
5. Reuven Bar-Yehuda and Shimon Even. A linear-time approximation algorithm for the weighted vertex cover problem. *Journal of Algorithms*, 2(2):198–203, 1981.
6. Reuven Bar-Yehuda and Shimon Even. A Local-Ratio Theorem for Approximating the Weighted Vertex Cover Problem. In *North-Holland Mathematics Studies*, volume 109, pages 27–45. Elsevier, 1985.
7. Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The locality of distributed symmetry breaking. *Journal of the ACM (JACM)*, 63(3):20, 2016.
8. Evelyn ML Beale. On minimizing a convex function subject to linear inequalities. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 173–184, 1955.
9. Ran Ben-Basat, Guy Even, Ken-ichi Kawarabayashi, and Gregory Schwartzman. A Deterministic Distributed 2-Approximation for Weighted Vertex Cover in O(\log N\log \varDelta /\log ^2\log \varDelta) Rounds. In *International Colloquium on Structural Information and Communication Complexity*, pages 226–236. Springer, 2018.
10. John R Birge and Francois Louveaux. *Introduction to stochastic programming*. Springer Science & Business Media, 2011.
11. Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.
12. George B Dantzig. LINEAR PROGRAMMING UNDER UNCERTAINTY. *Science*, 1955.

13 Kedar Dhamdhere, Vineet Goyal, R Ravi, and Mohit Singh. How to pay, come what may: Approximation algorithms for demand-robust covering problems. In *Foundations of Computer Science, 2005. FOCS 2005. 46th Annual IEEE Symposium on*, pages 367–376. IEEE, 2005.

14 Shane Dye, Leen Stougie, and Asgeir Tomasgard. The stochastic single resource service-provision problem. *Naval Research Logistics (NRL)*, 50(8):869–887, 2003.

15 Uriel Feige, Kamal Jain, Mohammad Mahdian, and Vahab Mirrokni. Robust combinatorial optimization with exponential scenarios. In *International Conference on Integer Programming and Combinatorial Optimization*, pages 439–453. Springer, 2007.

16 Manuela Fischer. Improved Deterministic Distributed Matching via Rounding. In *31st International Symposium on Distributed Computing, DISC*, pages 17:1–17:15, 2017.

17 Fabrizio Grandoni, Jochen Könemann, and Alessandro Panconesi. Distributed weighted vertex cover via maximal matchings. *ACM Transactions on Algorithms (TALG)*, 5(1):6, 2008.

18 Anupam Gupta, Martin Pál, R Ravi, and Amitabh Sinha. Sampling and cost-sharing: Approximation algorithms for stochastic optimization problems. *SIAM Journal on Computing*, 40(5):1361–1401, 2011.

19 Anupam Gupta, R Ravi, and Amitabh Sinha. LP rounding approximation algorithms for stochastic network design. *Mathematics of Operations Research*, 32(2):345–364, 2007.

20 Michal Hanckowiak, Michal Karonski, and Alessandro Panconesi. On the distributed complexity of computing maximal matchings. *SIAM Journal on Discrete Mathematics*, 15(1):41–57, 2001.

21 Nicole Immorlica, David Karger, Maria Minkoff, and Vahab S Mirrokni. On the costs and benefits of procrastination: Approximation algorithms for stochastic combinatorial optimization problems. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 691–700. Society for Industrial and Applied Mathematics, 2004.

22 Amos Israeli and Alon Itai. A Fast and Simple Randomized Parallel Algorithm for Maximal Matching. *Inf. Process. Lett.*, 22(2):77–80, 1986.

23 George Karakostas. A better approximation ratio for the vertex cover problem. *ACM Transactions on Algorithms (TALG)*, 5(4):41, 2009.

24 Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

25 Subhash Khot and Oded Regev. Vertex cover might be hard to approximate to within 2-epsilon. *J. Comput. Syst. Sci.*, 74(3):335–349, 2008.

26 Samir Khuller, Uzi Vishkin, and Neal Young. A primal-dual parallel approximation technique applied to weighted set and vertex covers. *Journal of Algorithms*, 17(2):280–289, 1994.

27 Willem K Klein Haneveld and Maarten H van der Vlerk. Stochastic integer programming: General models and algorithms. *Annals of Operations Research*, 85:39–57, 1999.

28 Christos Koufogiannakis and Neal E Young. Distributed algorithms for covering, packing and maximum weighted matching. *Distributed Computing*, 24(1):45–63, 2011.

29 Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. The price of being near-sighted. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 980–989. Society for Industrial and Applied Mathematics, 2006.

30 George L Nemhauser and Leslie Earl Trotter. Vertex packings: structural properties and algorithms. *Mathematical Programming*, 8(1):232–248, 1975.

31 Alessandro Panconesi and Romeo Rizzi. Some simple distributed algorithms for sparse networks. *Distributed computing*, 14(2):97–100, 2001.

32 David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. Discrete Mathematics and Applications. Society for Industrial and Applied Mathematics, 2000. URL: `https://books.google.co.il/books?id=T1hFWuDi1CsC`.

33 Valentin Polishchuk and Jukka Suomela. A simple local 3-approximation algorithm for vertex cover. *Information Processing Letters*, 109(12):642–645, 2009.

34 R Ravi and Amitabh Sinha. Hedging uncertainty: Approximation algorithms for stochastic optimization problems. *Mathematical Programming*, 108(1):97–114, 2006.

35 Rüdiger Schultz, Leen Stougie, and Maarten H van der Vlerk. Two-stage stochastic integer programming: a survey. *Statistica Neerlandica*, 50(3):404–416, 1996.

36 David B Shmoys and Chaitanya Swamy. An approximation scheme for stochastic linear programming and its application to stochastic integer programs. *Journal of the ACM (JACM)*, 53(6):978–1012, 2006.

37 Aravind Srinivasan. Approximation algorithms for stochastic and risk-averse optimization. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 1305–1313, 2007.

38 Chaitanya Swamy and David B Shmoys. Sampling-based approximation algorithms for multi-stage stochastic optimization. In *Foundations of Computer Science, 2005. FOCS 2005. 46th Annual IEEE Symposium on*, pages 357–366. IEEE, 2005.

39 Chaitanya Swamy and David B Shmoys. Approximation algorithms for 2-stage stochastic optimization problems. *ACM SIGACT News*, 37(1):33–46, 2006.

Equivalence Classes and Conditional Hardness in Massively Parallel Computations

Danupon Nanongkai
KTH Royal Institute of Technology, Sweden
danupon@gmail.com

Michele Scquizzato
University of Padova, Italy
scquizza@math.unipd.it

Abstract

The *Massively Parallel Computation* (MPC) model serves as a common abstraction of many modern large-scale data processing frameworks, and has been receiving increasingly more attention over the past few years, especially in the context of classical graph problems. So far, the only way to argue lower bounds for this model is to condition on conjectures about the hardness of some specific problems, such as graph connectivity on promise graphs that are either one cycle or two cycles, usually called the *one cycle vs. two cycles* problem. This is unlike the traditional arguments based on conjectures about complexity classes (e.g., $\mathsf{P} \neq \mathsf{NP}$), which are often more robust in the sense that refuting them would lead to groundbreaking algorithms for a whole bunch of problems.

In this paper we present connections between problems and classes of problems that allow the latter type of arguments. These connections concern the class of problems solvable in a sublogarithmic amount of rounds in the MPC model, denoted by $\mathsf{MPC}(o(\log N))$, and some standard classes concerning space complexity, namely L and NL, and suggest conjectures that are robust in the sense that refuting them would lead to many surprisingly fast new algorithms in the MPC model. We also obtain new conditional lower bounds, and prove new reductions and equivalences between problems in the MPC model.

2012 ACM Subject Classification Theory of computation → Massively parallel algorithms

Keywords and phrases Massively parallel computation, conditional hardness, fine-grained complexity

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.33

Funding This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme under grant agreement No 715672. M. Scquizzato was also partially supported by the University of Padova under grant BIRD197859/19.

1 Introduction

The *Massively Parallel Computation* (MPC) model is arguably the most popular model of computation that captures the essence of several very successful general-purpose frameworks for massively parallel coarse-grained computations on large data sets, such as MapReduce [23], Hadoop [60], Spark [62], and Dryad [38]. The MPC model, introduced by Karloff et al. [40], and originally inspired by the MapReduce paradigm, aims at modeling distributed-memory parallel computations in the situation when the size of the input is so big that a single machine cannot even store the whole input, but just a strongly sublinear fraction of it. The computation proceeds in synchronous rounds, and in each of them the machines can exchange data with each other with the sole restriction that no one can ever receive more data than it is capable of storing. The goal is to keep the total number of rounds as low as possible.

© Danupon Nanongkai and Michele Scquizzato;
licensed under Creative Commons License CC-BY
23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller; Article No. 33; pp. 33:1–33:16
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

This basic model has been much investigated in the past decade, mostly from an algorithmic point of view [40, 45, 33, 12, 52, 53, 2, 43, 35, 4, 39, 26, 13, 37, 55, 15, 22, 61, 29, 5, 11, 9, 8, 31, 34, 28, 36, 6, 14, 20, 17, 30, 16]. A common outcome is that, when N denotes the input size, a solution terminating in $O(\log N)$ rounds is possible, usually by simulating known PRAM algorithms [40, 33], but going below that resisted the efforts of many researchers. Recently, a few works managed to break the $O(\log N)$ barrier by relaxing a bit the strongly-sublinear constraint on the memory size, and showed that some graph problems allow for $o(\log N)$-round solutions in the so-called *near-linear memory* regime, whereby machines have memories of size $\tilde{O}(n)$, where n is the number of nodes in the graph [22, 29, 8, 9, 17]. However, without this kind of relaxations only a handful of problems are known to admit a $o(\log N)$-round algorithm [31, 34, 20]. A fundamental question is thus whether many known $O(\log N)$-round algorithms can be complemented with tight lower bounds.

Unfortunately, proving *unconditional* lower bounds – that is, without any assumptions – seems extremely difficult in this model, as it would imply a breakthrough in circuit complexity: Roughgarden et al. [55] showed that, when enough machines are available, proving any super-constant lower bound for any problem in P would imply new circuit lower bounds, and specifically would separate NC^1 from P – a long-standing open question in complexity theory that is a whisker away from the P vs. NP question. This means that the lack of super-constant lower bounds in the MPC model can be blamed on our inability to prove some computational hardness results.

In light of this barrier, the focus rapidly shifted to proving *conditional* lower bounds, that is, lower bounds conditioned on plausible hardness assumptions. One widely-believed assumption concerns graph connectivity, which, when machines have a memory of size $O(n^{1-\epsilon})$ for some constant $\epsilon > 0$, is conjectured to require $\Omega(\log n)$ MPC rounds [40, 53, 13, 55, 61].[1] The same conjecture is often made even for the special case of the problem where the graph consists of either one cycle or two cycles, usually called *one cycle vs. two cycles* problem. The one cycle vs. two cycles conjecture has been proven useful to show conditional lower bounds for some problems, such as minimum spanning trees in low-dimensional spaces [4], single-linkage clustering [61], 2-vertex connectivity [6], generation of random walks [44], as well as parameterized conditional lower bounds [16].[2]

However, it is not clear whether the one cycle vs. two cycles conjecture is true or not, and if not, what its refutation implies. This situation is in contrast with traditional complexity theory, where a refutation of a conjectured relationship between complexity classes would typically imply groundbreaking algorithmic results for a large number of problems; for example, if the $\mathsf{P} \neq \mathsf{NP}$ conjecture fails, then there would be efficient (polynomial-time) algorithms for *all* problems in NP, including a number of "hard" problems. To put it another way, a conjecture like $\mathsf{P} \neq \mathsf{NP}$ is more *robust* in the sense that it is extremely hard to refute – doing so requires a major algorithmic breakthrough. The goal of this paper is to explore conjectures of this nature in the MPC model.

[1] Observe that in the near-linear memory regime this conjecture breaks: graph connectivity can be solved in $O(1)$ MPC rounds [15].

[2] The one cycle vs. two cycles problem is usually stated such that, in the case of two cycles, these have $n/2$ nodes each. However, we observe that all the mentioned conditional lower bounds hold also when the two cycles may have arbitrary lengths.

1.1 Summary of Contributions

In this paper we show many connections between problems and classes of problems that lead to more robust conjectures for the MPC model. In particular, we study the connections between the class of problems solvable in a sublogarithmic amount of rounds in the MPC model, denoted by $\mathsf{MPC}(o(\log N))$, and the standard space complexity classes L and NL. (Recall that L and NL are the classes of decision problems decidable in logarithmic space on deterministic and nondeterministic Turing machines, respectively.) The connection between MPC and these complexity classes is enabled by a recent result showing how Boolean circuits can be efficiently simulated in the MPC model. In short, we present a set of observations and reductions that suggest that $\mathsf{L} \not\subseteq \mathsf{MPC}(o(\log N))$ and $\mathsf{NL} \not\subseteq \mathsf{MPC}(o(\log N))$ are two robust conjectures that might play crucial roles in arguing lower bounds in the MPC model, as they already imply tight conditional lower bounds for a large number of problems. In particular, with some assumptions on the total amount of memory (equivalently, machines) available in the system, we can conclude the following.

1. **Robustness:** The one cycle vs. two cycles conjecture is robust, since it is equivalent to conjecturing that $\mathsf{L} \not\subseteq \mathsf{MPC}(o(\log N))$, and refuting this conjecture requires showing $o(\log N)$-round algorithms for all problems in L. This class includes many important problems such as graph connectivity, cycle detection, and planarity testing (see problems in the bottom ellipse in Figure 1 for more).
2. **Equivalences:** All L-complete problems are *equivalent* in the sense that they require asymptotically the same number of rounds. This means that the one cycle vs. two cycles problem, which is L-complete (see Appendix A), is equivalent to many seemingly harder problems, such as graph bipartiteness, minimum cut, and formula evaluation (see problems in the bottom ellipse in Figure 1 for more).

   Additionally, all NL-complete problems and a few others are also equivalent. These problems include *st*-reachability, all-pairs shortest paths (both the directed and undirected cases) on unweighted graphs, diameter, and betweenness centrality (see problems in the top ellipse in Figure 1 for more).
3. **New conditional lower bounds:** Assuming the one cycle vs. two cycles conjecture (equivalently, $\mathsf{L} \not\subseteq \mathsf{MPC}(o(\log N))$), there are no $o(\log N)$-round algorithms for all L-hard problems and a few other problems. This implies new conditional lower bounds for more than a dozen of problems, such as betweenness centrality, planarity testing, graph bipartiteness, list ranking, formula evaluation, and densest subgraph (see problems in the big rectangle in Figure 1 for more). Previously only a few lower bounds were known, e.g., those for single-linkage clustering [61] and maximum matching [48]. (Of course, lower bounds for connectivity-related problems are trivially implied by the one cycle vs. two cycles conjecture.) Most of our lower bounds are tight (e.g., lower bounds for problems in the ellipses in Figure 1).
4. **A more robust conjecture.** For NL-hard problems, we can argue lower bounds under the more robust $\mathsf{NL} \not\subseteq \mathsf{MPC}(o(\log N))$ conjecture. These problems include perfect matching, single-source shortest paths, diameter, and network flow (see problems in the small rectangle in Figure 1 for more). Note that the $\mathsf{NL} \not\subseteq \mathsf{MPC}(o(\log N))$ conjecture is more robust (i.e., more likely to be true) since $\mathsf{L} \subseteq \mathsf{NL}$.

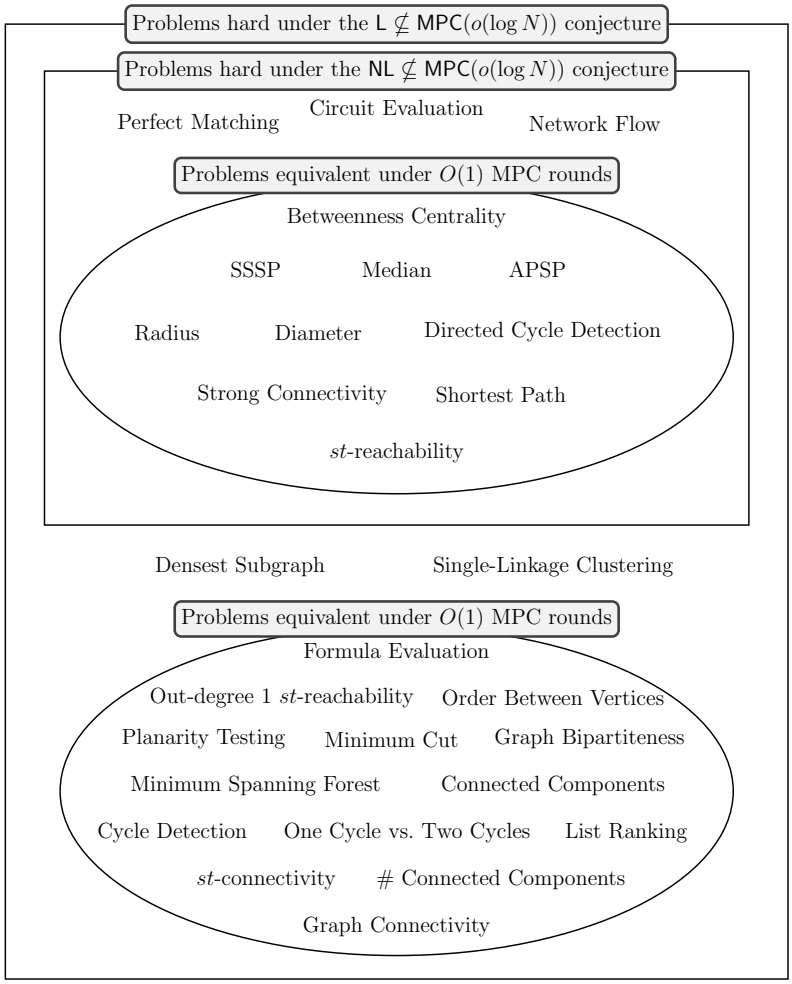


Figure 1 A classification of the complexity of some prominent problems in the MPC model. Problems in the top ellipse are on unweighted graphs.

1.2 Related Work

Fish et al. [26] were perhaps the first to establish a connection between the MPC model and classical complexity classes. Besides the introduction of a uniform version of the model, they showed that constant-round MPC computations can simulate sublogarithmic space-bounded Turing machines, and then proved strict hierarchy theorems for the MPC model under certain complexity-theoretic assumptions.

Roughgarden et al. [55] discuss connections between the MPC model and Boolean circuits. They show that standard degree arguments for circuits can be applied to MPC computations as well, and specifically that any Boolean function whose polynomial representation has degree d requires $\Omega(\log_s d)$ rounds of MPC using machines with memory s. This implies an $\Omega(\log_s n)$ lower bound on the number of rounds for graph connectivity. Perhaps more interestingly, the authors show a barrier for unconditional lower bounds by observing that, if enough machines are available, then proving any super-constant lower bound in the MPC model for any problem in P would imply new circuit lower bounds, and specifically would

separate NC^1 from P, thus answering a notorious open question in circuit complexity. This result follows by showing that, with a number of available machines polynomial in the number of input nodes of the circuit, NC^1 circuits can be efficiently simulated in the MPC model. We observe that their argument readily generalizes to show that any bounded fan-in Boolean circuit of depth d and of polynomial size can be simulated in $O(\lceil d/\log s\rceil)$ MPC rounds. Very recently, Frei and Wada [27] prove the same result improving over the amount of machines required for the simulation – from linear to strongly sublinear in the size of the circuit.

Given the difficulty of proving lower bounds for all algorithms, one can (a) prove lower bounds for restricted classes of algorithms, or (b) prove conditional lower bounds: assume one lower bound, and transfer the conjectured hardness to other problems via reductions (with common examples being the theory of NP-hardness and its more recent analogue for problems in P, usually called *fine-grained complexity theory*). Both paths give a deep understanding and warn us what not to try when designing algorithms.

Within the first line of inquiry, Pietracaprina et al. [52] prove lower bounds for matrix multiplication algorithms that compute all the n^3 elementary products. Similar kinds of limitations are required by Beame et al. [13], Jacob et al. [39], Im and Moseley [36], and Assadi and Khanna [10] to prove lower bounds for st-connectivity, list ranking, graph connectivity, and maximum coverage, respectively. Of a similar flavor are the results of Afrati et al. [2], who show, for a fixed number of rounds (usually a single round), space-communication tradeoffs.

Within the second line of inquiry fall [4, 61, 6, 44], which use the conjecture on the hardness of graph connectivity as a hardness assumption for proving conditional lower bounds for other problems such as minimum spanning trees in low-dimensional spaces, single-linkage clustering, 2-vertex connectivity, and generating random walks, respectively. Very recently, Ghaffari et al. [30] present conditional lower bounds for other key graph problems such as approximate maximum matching, approximate vertex cover, maximal independent set, and maximal matching. Their lower bounds also rest on the hardness of graph connectivity, and are obtained by introducing a new general method that lifts (unconditional) lower bounds from the classical LOCAL model of distributed computing to the MPC model. A conditional lower bound following a different kind of argument is given by Andoni et al. [5], who show that an $n^{o(1)}$-round MPC algorithm that answers $O(n+m)$ pairs of reachability queries in directed graphs with n nodes and m edges can be simulated in the RAM model yielding faster Boolean matrix multiplication algorithms.

Several other models have been developed in the quest to establish rigorous theoretical foundations of (massively) parallel computing, with the PRAM being one of the most investigated. The MPC model is more powerful than the PRAM since PRAM algorithms can be simulated in the MPC model with constant slowdown [40, 33], and some problems (such as evaluating the XOR function) can be solved much faster in the MPC model.

Valiant's *bulk-synchronous parallel* (BSP) model [58] anticipated many of the features of MPC-type computations, such as the organization of the computation in a sequence of synchronous rounds (originally called *supersteps*). Several papers (e.g., [32, 47, 1, 56, 18]) explored the power of this model by establishing lower bounds on the number of supersteps or on the communication complexity required by BSP computations. Lower bounds on the number of supersteps are usually of the form $\Omega(\log_h N)$, where h is the maximum number of messages sent or received by any processor in any superstep.

Another model aiming at serving as an abstraction for modern large-scale data processing frameworks is the *k-machine* model [41]. Partly inspired by message-passing models in distributed computing, in the k-machine model there are k available machines, and in each

round any pair of machines is allowed to communicate using messages of a given size. Hard bounds on the point-to-point communication lead to very strong round lower bounds in this model [41, 49, 50].

The *congested clique* (see, e.g., [24]) is a model for network computations bearing some similarities with the MPC model. On one hand, algorithms for this model can be simulated in the MPC model – under some specific conditions on the size of the local memories [35, 29, 15]. On the other hand, analogously to the MPC model, proving a super-constant unconditional lower bound in the congested clique for a problem in NP would imply better circuit size-depth tradeoffs for such a problem than are currently known [24]. This induced further investigations of the model under the lens of complexity theory [42].

2 Preliminaries

In this section we present the MPC model in detail. We assume that the reader is familiar with the standard space complexity classes L and NL, and with the logspace-uniform circuit complexity classes NC^k and AC^k (see, e.g., the textbook [7]).

2.1 The MPC Model

The *Massively Parallel Computation (MPC)* model is a theoretical abstraction capturing the main distinguishing aspects of several popular frameworks for the parallel processing of large-scale datasets. It was introduced by Karloff, Suri, and Vassilvitskii [40], and refined in subsequent work [33, 13, 4].

In this model the system consists of p identical machines (processors), each with a local memory of size s. If N denotes the size of the input, then $s = O(N^{1-\epsilon})$ for some fixed constant $\epsilon > 0$, and the total amount of memory available in the system is $p \cdot s = O(N^{1+\gamma})$ for some fixed constant $\gamma \geq 0$. The space size is measured by words, each of $\Theta(\log N)$ bits. Initially, the input is adversarially distributed across the machines. The computation proceeds in synchronous rounds. In each round, each machine performs some computation on the data that resides in its local memory, and then, at the end of the round, exchanges messages with other machines. The total size of messages sent or received by each machine in each round is bounded by s. The goal is to minimize the total number of rounds.

For problems defined on graphs, the input size N is equal to $n+m$, where n is the number of nodes of the graph and m is the number of edges. When considering graph problems, in this paper we assume $s = O(n^{1-\epsilon})$. This regime of memory size, usually called *strongly sublinear memory* regime, is always in compliance with the aforementioned constraint on the size of the local memory, even when graphs are sparse, for which the constraint is the most restrictive.

Since we want to relate the MPC model to classical complexity classes, one must make sure that the model is *uniform*, by which we mean, roughly speaking, that the same algorithm solves the problem for inputs of all (infinitely many) sizes. Fish et al. [26] dealt with this issue observing that Karloff et al.'s original definition of the model [40] is non-uniform, allowing it to decide undecidable languages, and thus by reformulating the definition of the model to make it uniform. Building on that reformulation, and letting $f\colon \mathbb{N} \to \mathbb{R}^+$ be a function, we define the class $\mathsf{MPC}(f(N))$ to be the class of problems solvable in $O(f(N))$ MPC rounds by a uniform family of MPC computations.

3 Massively Parallel Computations and Space Complexity Classes

In this section we recall a recent result showing that Boolean circuits can be efficiently simulated in the MPC model, and then we build on it to derive new results and conjectures.

3.1 Efficient Circuit Simulation in the MPC Model

We now recall the main result in [27] which, roughly speaking, says that any bounded fan-in Boolean circuit of depth d and of polynomial size can be simulated in $O(\lceil d/\log s\rceil)$ MPC rounds. This result is already implicit in [55], where it is achieved by a simple simulation whereby each gate of the circuit is associated with a machine whose responsibility is to compute the output of the gate. This requires the availability of a number of machines polynomial in the number of inputs n of the circuit, and thus linear in the size of the circuit. Very recently, Frei and Wada [27] came up with a more sophisticated strategy, which uses only a strongly sublinear amount of machines. Their strategy employs two distinct simulations: for NC^1 circuits they exploit Barrington's well-known characterization of NC^1 in terms of bounded-width polynomial-size branching programs, and thus simulate such branching programs in a constant number of rounds; for the higher levels of the NC hierarchy, the Boolean circuits themselves are directly simulated, suitably dividing the computation into the simulation of sub-circuits of depth $O(\log n)$, each to be accomplished in $O(1)$ rounds.

The authors work in the original model of Karloff et al. [40], but their result seamlessly applies in the refined MPC model.

▶ **Theorem 1** ([27]). *Let DMPC^i denote the class of problems solvable by a deterministic MPC algorithm in $O(\log^i N)$ rounds with $O(N^{1-\epsilon})$ local memory per machine and $O(N^{2(1-\epsilon)})$ total memory. Then,*

$$\mathsf{NC}^{i+1} \subseteq \mathsf{DMPC}^i$$

for every $i \in \mathcal{N}$ and for every $\epsilon \in (0, 1/2)$. (When $i = 0$, the result holds also for $\epsilon = 1/2$.)

Setting $i = 0$, we have the following.

▶ **Corollary 2.** *The class NC^1 can be simulated in $O(1)$ MPC rounds with $O(N^{1-\epsilon})$ local memory per machine and $O(N^{2(1-\epsilon)})$ total memory, for any constant $\epsilon \in (0, 1/2]$.*

Since $\mathsf{NC}^1 \subseteq \mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{NC}^2$ (see, e.g., [51]), an immediate by-product of Theorem 1 is that some standard space complexity classes can be efficiently simulated in the MPC model.

▶ **Corollary 3.** *The class NC^2, and thus the classes L and NL, can be simulated in $O(\log N)$ MPC rounds with $O(N^{1-\epsilon})$ local memory per machine and $O(N^{2(1-\epsilon)})$ total memory, for any constant $\epsilon \in (0, 1/2)$.*

3.2 New Consequences of Circuit Simulations

In this section we discuss new consequences of the fact that the MPC model is powerful enough to efficiently simulate general classes of Boolean circuits.

▶ **Theorem 4.** *Consider the MPC model where the size of the local memory per machine is $O(N^{1-\epsilon})$ for any constant $\epsilon \in (0, 1/2]$, and assume that $\Omega(N^{2(1-\epsilon)})$ total memory is available. Let $f\colon \mathbb{N} \to \mathbb{R}^+$ be a function. Then, if any L-hard problem can be solved in $O(f(N))$ MPC rounds, so can all the problems in the class L. Moreover, either all L-complete problems can be solved in $O(f(N))$ MPC rounds, or none of them can.*

Proof. Both claims follow directly from the definitions of L-hardness and L-completeness, and from Corollary 2. Let A be an L-hard problem that can be solved in $O(f(N))$ MPC rounds. By definition of L-hardness, every problem in L is NC^1 reducible to A. By assumption, $\epsilon \in (0, 1/2]$ and $\Omega(N^{2(1-\epsilon)})$ total memory is available, and thus, by Corollary 2, an NC^1 reduction can be simulated in $O(1)$ MPC rounds, giving the first claim. Therefore, in particular, if any L-complete problem can be solved in $O(f(N))$ MPC rounds, so can all the other L-complete problems. In other words, either all L-complete problems can be solved in $O(f(N))$ MPC rounds, or none of them can. ◀

We remark that in Theorem 4 no assumption is placed on the function $f(N)$, which therefore can be of any form, even a constant. Hence, Theorem 4 says that all the known L-complete problems such as graph connectivity, graph bipartiteness, cycle detection, and formula evaluation, are *equivalent* in the MPC model, and in a very strong sense: they all require asymptotically the same number of rounds. (Analogous equivalences are common in computer science, e.g., in fine-grained complexity theory, where equivalence classes of problems within P, such as the APSP class [59], are established.) Thus, this simple result provides an explanation of the striking phenomenon that for these well-studied problems we seem unable to break the $O(\log N)$ barrier in the MPC model. It also implies that the conjectures on the hardness of graph connectivity and on the hardness of the one cycle vs. two cycles problem are equivalent, at least when $\Omega(N^{2(1-\epsilon)})$ total memory is available.

The next theorem provides an even stronger barrier for improvements in the MPC model.

▶ **Theorem 5.** *Consider the MPC model where the size of the local memory per machine is $O(N^{1-\epsilon})$ for any constant $\epsilon \in (0, 1/2]$, and assume that $\Omega(N^{2(1-\epsilon)})$ total memory is available. Let $f : \mathbb{N} \to \mathbb{R}^+$ be a function. If any L-hard problem can be solved in $O(f(N))$ MPC rounds, then either all NL-complete problems can be solved in $O(f(N))$ MPC rounds, or none of them can. Moreover, if any NL-hard and any L-hard problem can be solved in $O(f(N))$ MPC rounds, so can all the problems in the class NL.*

Proof. Let A be an L-hard problem that can be solved in $O(f(N))$ MPC rounds. Then, by Theorem 4, every problem in the class L can be solved in $O(f(N))$ MPC rounds and thus, in particular, every log-space reduction can be computed in $O(f(N))$ MPC rounds. By definition of NL-completeness, every problem in NL, and thus, in particular, any NL-complete problem, is log-space reducible to any other NL-complete problem, and this proves the first statement.

Let B be an NL-hard problem that can be solved in $O(f(N))$ MPC rounds. By definition of NL-hardness, every problem in NL is log-space reducible to B. Since we have just argued that if any L-hard problem can be solved in $O(f(N))$ MPC rounds, so can any log-space reduction, the second statement follows. ◀

Once again, we stress that in Theorem 5 no assumption is placed on the function $f(N)$, which therefore can be of any form, even a constant.

Theorem 5 indicates that, unless $\mathsf{L} = \mathsf{NL}$, in the MPC model the connectivity problem on directed graphs, which is both NL-complete and L-hard, is strictly harder than on undirected graphs in the sense that breaking the current logarithmic barrier, if possible, would be strictly harder.

3.2.1 New Conjectures

The common belief that problems such as graph connectivity and list ranking cannot be solved in $o(\log N)$ MPC rounds, along with the equivalence result of Theorem 4, justify the following conjecture.

▶ **Conjecture 1.** *No L-hard problem can be solved in $o(\log N)$ MPC rounds with $O(N^{1-\epsilon})$ local memory per machine, for any constant $\epsilon \in (0,1)$, not even with a polynomial amount of total memory. Equivalently,*

$$\mathsf{L} \not\subseteq \mathsf{MPC}(o(\log N)).$$

We now show the claimed equivalence.

▶ **Proposition 6.** *The two statements in Conjecture 1 are equivalent.*

Proof. We shall argue that if any of the two statements is wrong, so is the other, and vice versa. Assume $\mathsf{L} \subseteq \mathsf{MPC}(o(\log N))$. Then, some L-complete, and hence L-hard, problem is contained in $\mathsf{MPC}(o(\log N))$, that is, it can be solved in $o(\log N)$ MPC rounds. To show the other direction, assume that there exists an L-hard problem that can be solved in $o(\log N)$ MPC rounds with a polynomial amount of total memory. Then, by Theorem 4, every problem in L can be solved in $o(\log N)$ MPC rounds, i.e., $\mathsf{L} \subseteq \mathsf{MPC}(o(\log N))$. ◀

We would like to remark that, in light of Theorem 4, Conjecture 1 is totally equivalent to the preceding conjectures on the hardness of graph connectivity or of the one cycle vs. two cycles problem [40, 53, 13, 55, 61]; however, Theorem 4 significantly strengthens the evidence for such conjectures.

Likewise, Theorem 5 provides a justification for the following conjecture.

▶ **Conjecture 2.** *No NL-hard and L-hard problem can be solved in $o(\log N)$ MPC rounds with $O(N^{1-\epsilon})$ local memory per machine, for any constant $\epsilon \in (0,1)$, not even with a polynomial amount of total memory. Equivalently,*

$$\mathsf{NL} \not\subseteq \mathsf{MPC}(o(\log N)).$$

The two statements in Conjecture 2 are equivalent. The argument is similar to that used for the preceding proposition.

Observe that since $\mathsf{L} \subseteq \mathsf{NL}$, Conjecture 1 implies Conjecture 2. Hence, unless $\mathsf{L} = \mathsf{NL}$, Conjecture 2 is weaker than Conjecture 1, and thus more likely to be true.

4 Reductions and Equivalences in Massively Parallel Computations

In this section we discuss two equivalence classes of problems and some conditional lower bounds in the MPC model. The two equivalence classes both contain problems equivalent to each other under $O(1)$-round MPC reductions and for which the best known upper bound is $O(\log N)$ rounds, but differ in terms of the low-space computational complexity characterization of the problems they contain.

As a consequence of the results of Section 3, most of these reductions and equivalences follow from known hardness and completeness results for low-space complexity classes such as L and NL.

We will also show novel reductions and equivalences in the MPC model. Some of such reductions crucially require the availability of up to polynomially many machines (equivalently, a total amount of memory up to polynomial in the input size), which are used to host up to a polynomial number of copies of the input data. The quick creation of so many input replicas can be achieved through the use of a simple two-step broadcast procedure, as shown in the following lemma.

▶ **Lemma 7.** *The input data can be replicated up to a polynomial number of times in $O(1)$ MPC rounds.*

4.1 An Equivalence Class for Undirected Graph Connectivity

In this section we discuss the MPC equivalence class for graph connectivity in undirected graphs. This problem, which asks to determine whether a given undirected graph is connected or not, was one of the first problems to be shown L-hard under (uniform) NC^1 reductions [21], and then it was placed in L by the remarkable algorithm of Reingold [54]. Exploiting the results of Section 3, we know that one can recycle all the reductions that have been developed in classical complexity theory for showing hardness and completeness for class L in the MPC model as well, since these can all be simulated in $O(1)$ MPC rounds with $O(N^{2(1-\epsilon)})$ total memory. This immediately implies that the class of L-complete problems forms an equivalence class in the MPC model as well. Specifically, for example, either all the following problems can be solved with a sublogarithmic MPC algorithm, or none of them can:graph connectivity, connectivity for promise graphs that are a disjoint union of cycles, *st*-connectivity, *st*-reachability for directed graphs of out-degree one, cycle detection, order between vertices, formula evaluation, and planarity testing.

Recycling (some) old SL-completeness results. Many more problems can be placed in this MPC equivalence class almost effortlessly: this is the case for some problems complete for the class *symmetric logarithmic space* (SL), a class defined by Lewis and Papadimitriou [46] to capture the complexity of undirected *st*-connectivity before this was eventually settled by the breakthrough of Reingold. Completeness in SL is defined in terms of log-space reductions, and *st*-connectivity is one complete problem for it. Since $\mathsf{L} \subseteq \mathsf{SL}$, Reingold's algorithm made these two classes collapse, thus widening the class L with many new problems. However, completeness for SL does not translate into completeness for L, since the latter is defined in terms of a lower-level kind of reduction. Luckily, some of the log-space reductions devised to show hardness for SL turn out to be actually stronger than log-space. This is the case, e.g., of testing whether a given graph is bipartite (or, equivalently, 2-colorable), as we show next.

▶ **Lemma 8.** *Graph bipartiteness is equivalent to st-connectivity under $O(1)$-round MPC reductions, with $O(n^{1-\epsilon})$ local memory per machine for any constant $\epsilon \in (0,1)$, and $O(n(n+m))$ total memory.*

A good source of problems complete for SL is [3].

From decision to non-decision problems. Complexity classes such as L contain problems phrased as decision problems. Nevertheless, it is often easy to transform them into their non-decision version. As an example, consider *order between vertices* (ORD). ORD is the decision version of list ranking, the problem of obtaining a total ordering from a given successor relation [25]. It is easy to argue the following equivalence.

▶ **Lemma 9.** *List ranking is equivalent to order between vertices under $O(1)$-round MPC reductions, with $O(n^{1-\epsilon})$ local memory per machine for any constant $\epsilon \in (0,1)$, and $O(n^3)$ total memory.*

Non-pairwise reductions. Sometimes back-and-forth reductions between two problems are not known. In this case their equivalence may nevertheless be established through a series of reductions involving related problems.

▶ **Lemma 10.** *Graph connectivity, st-connectivity, # connected components, connected components, minimum spanning forest, and minimum cut are all equivalent under $O(1)$-round MPC reductions, with $O(n^{1-\epsilon})$ local memory per machine for any constant $\epsilon \in (0,1)$, and $\tilde{O}(n^2 m(n+m))$ total memory.*

We can now summarize all the results of this section.

▶ **Theorem 11.** *The following problems are all equivalent under $O(1)$-round MPC reductions, with $O(n^{1-\epsilon})$ local memory per machine for any constant $\epsilon \in (0,1)$, and $\tilde{O}(n^2 m(n+m))$ total memory: graph connectivity, connectivity for promise graphs that are a disjoint union of cycles, st-connectivity, st-reachability for directed graphs of out-degree one, cycle detection, order between vertices, formula evaluation, planarity testing, graph bipartiteness, list ranking, # connected components, connected components, minimum spanning forest, and minimum cut.*

Conditional hardness: L-hard problems. Finally, there are problems known to be L-hard, but not known to be in L, such as densest subgraph and perfect matching. Since for these problems only one-way reductions from problems in L are known, we don't know whether they are part of the equivalence class of undirected graph connectivity.

4.2 An Equivalence Class for Directed Graph Connectivity

In this section we discuss the MPC equivalence class for graph connectivity in directed graphs. The problem corresponding to st-connectivity in directed graphs is *st-reachability*, that is, the problem of detecting whether there is a path from a distinguished node s to a distinguished node t in a directed graph. st-reachability is the prototypical complete problem for NL [51, 57, 7].

Recall that hardness in NL is defined with respect to log-space reducibility, but we do not know whether log-space computations can be simulated in $o(\log N)$ MPC rounds – in fact, in Section 3 we conjecture they cannot. However, it turns out that many of the known log-space reductions that establish NL-hardness of problems can actually be simulated in $O(1)$ MPC rounds. This is the case, for example, of the reductions between st-reachability and *shortest path*, the other canonical example NL-complete problem which, given an undirected (unweighted) graph, two distinguished nodes s and t, and an integer k, asks to determine if the length of a shortest path from s to t is k.

▶ **Lemma 12.** *Shortest path on unweighted graphs is equivalent to st-reachability under $O(1)$-round MPC reductions, with $O(n^{1-\epsilon})$ local memory per machine for any constant $\epsilon \in (0,1)$, and $O(n^{\delta}(n+m))$ total memory where δ is a small enough positive constant.*

There are other NL-complete problems that can be shown to be equivalent under $O(1)$-round MPC reductions. Some examples are directed cycle detection, by a simple adaptation of the preceding reductions, and strong connectivity, which follows from a result in [19]. We suspect that many other log-space reductions are actually (or can easily be translated into) $O(1)$-round MPC reductions, thus enabling us to enlarge the equivalence class for graph connectivity in directed graphs almost effortlessly by leveraging known results in complexity theory.

When this is not possible, one might have to devise novel reductions. We now do so for some important shortest-path-related problems as well as for some graph centrality problems.

4.2.1 New Fine-Grained MPC Reductions: Constant-Round Equivalences Between Graph Centrality Problems, APSP, and Diameter

In this section we shall exploit the *shortest path* problem as a prototypical problem for which the fastest known MPC algorithm takes $O(\log n)$ rounds, and prove a collection of constant-round equivalences with many other graph problems.

We start by showing the simple fine-grained equivalence between APSP and shortest path.

▶ **Lemma 13.** *APSP is equivalent to shortest path under $O(1)$-round MPC reductions, with $O(n^{1-\epsilon})$ local memory per machine for any constant $\epsilon \in (0,1)$, and $O(n^2(n+m))$ total memory.*

Proof. The reduction from shortest path to APSP is obvious. The other direction is also immediate when we have enough machines, and specifically $O(n^2(n+m))$ total memory: by Lemma 7 we can create $2\binom{n}{2}$ copies of the input graph in $O(1)$ MPC rounds, and then in parallel, one pair for each copy, compute the shortest path for each (ordered, if the graph is directed) pair of nodes. ◀

In the following results we will use roughly the same reduction. We start with the problem of determining the diameter of a graph.

▶ **Lemma 14.** *Shortest path is $O(1)$-round MPC reducible to diameter, with $O(n^{1-\epsilon})$ local memory per machine for any constant $\epsilon \in (0,1)$, and $O(n+m)$ total memory.*

Proof. We start with the case of undirected graphs. Given an instance of shortest path, the idea is to alter the input graph by sticking two new and sufficiently long paths to nodes s and t, so that the path of largest total weight includes both s and t.

This is sufficient if the original graph G is connected; otherwise, the diameter is infinite, and from this information we cannot determine the length of a shortest path from s to t. Hence, we shall first make G connected in a way that alters the distance between s and t only if they are not connected in G. Since the distance between any two nodes can be at most $(n-1)M$, this can be achieved by adding to the graph a new node v and n edges of weight nM between v and any other node. Then, we append two additional chains to s and t, each with $2n$ edges of weight M, and denote this modified graph by G'.

This reduction can be performed in $O(1)$ MPC rounds, it increases the number of nodes and the number of edges by $O(n)$, and the maximum absolute weight by a factor of $O(n)$. Therefore, any MPC algorithms that runs in $O(T(n,m,M))$ rounds in the new graph G' can be used to solve the original instance G in $O(T(n,m,M))$ rounds as well.

Observe that the diameter of the modified graph G' must include the two chains appended to s and t. Hence any algorithm for the diameter when executed on graph G' always returns $4nM$ plus the shortest-path distance between s and t in G'. By construction, the latter quantity, which we denote by α, is at most $(n-1)M$ if s and t are connected in G, and (exactly) $2nM$ otherwise. Thus the answer to shortest path is α if the diameter of G' is at most $4nM + (n-1)M$, and infinity otherwise.

In the directed case, we use the same weighted graph G' as before, adding one parallel edge for each edge, both with the same weight but with opposite directions. The rest of the algorithm is the same and its analysis is analogous to the undirected case. ◀

▶ **Lemma 15.** *Shortest path is $O(1)$-round MPC reducible to radius, with $O(n^{1-\epsilon})$ local memory per machine for any constant $\epsilon \in (0,1)$, and $O(n+m)$ total memory.*

▶ **Lemma 16.** *Shortest path is $O(1)$-round MPC reducible to median, with $O(n^{1-\epsilon})$ local memory per machine for any constant $\epsilon \in (0,1)$, and $O(n+m)$ total memory.*

Now we consider the evaluation of the betweenness centrality of nodes. In contrast to the previous reductions, in the following one we shall create n copies of the reduction graph leveraging Lemma 7, and then perform some computation in parallel.

▶ **Lemma 17.** *Shortest path is $O(1)$-round MPC reducible to betweenness centrality, with $O(n^{1-\epsilon})$ local memory per machine for any constant $\epsilon \in (0,1)$, and $O(n(n+m))$ total memory.*

An immediate consequence of these results is the following.

▶ **Proposition 18.** *Shortest path, SSSP, APSP, diameter, radius, median, and betweenness centrality are all equivalent under $O(1)$-round MPC reductions, with $O(n^{1-\epsilon})$ local memory per machine for any constant $\epsilon \in (0,1)$, and $O(n^2(n+m))$ total memory.*

Proof. The two reductions involving SSSP are obvious. The reduction from diameter (or radius) to APSP is also obvious, since determining the maximum (or minimum) in a set of values can be easily done in $O(1)$ MPC rounds. The theorem then follows from Lemmas 13 to 17. ◀

We can now summarize all the results of this section.

▶ **Theorem 19.** *The following problems are all equivalent under $O(1)$-round MPC reductions, with $O(n^{1-\epsilon})$ local memory per machine for any constant $\epsilon \in (0,1)$, and $O(n^2(n+m))$ total memory: st-reachability, strong connectivity, directed cycle detection, unweighted shortest path, unweighted SSSP, unweighted APSP, unweighted diameter, unweighted radius, unweighted median, and unweighted betweenness centrality.*

References

1. Micah Adler, Wolfgang Dittrich, Ben H. H. Juurlink, Miroslaw Kutylowski, and Ingo Rieping. Communication-optimal parallel minimum spanning tree algorithms. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 27–36, 1998.
2. Foto N. Afrati, Anish Das Sarma, Semih Salihoglu, and Jeffrey D. Ullman. Upper and Lower Bounds on the Cost of a Map-Reduce Computation. *PVLDB*, 6(4):277–288, 2013.
3. Carme Àlvarez and Raymond Greenlaw. A compendium of problems complete for symmetric logarithmic space. *Computational Complexity*, 9(2):123–145, 2000.
4. Alexandr Andoni, Aleksandar Nikolov, Krzysztof Onak, and Grigory Yaroslavtsev. Parallel algorithms for geometric graph problems. In *Proceedings of the 46th Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 574–583, 2014.
5. Alexandr Andoni, Clifford Stein, Zhao Song, Zhengyu Wang, and Peilin Zhong. Parallel Graph Connectivity in Log Diameter Rounds. In *Proceedings of the 59th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 674–685, 2018.
6. Alexandr Andoni, Clifford Stein, and Peilin Zhong. Log Diameter Rounds Algorithms for 2-Vertex and 2-Edge Connectivity. In *Proceedings of the 46th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 14:1–14:16, 2019.
7. Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.
8. Sepehr Assadi, MohammadHossein Bateni, Aaron Bernstein, Vahab S. Mirrokni, and Cliff Stein. Coresets Meet EDCS: Algorithms for Matching and Vertex Cover on Massive Graphs. In *Proceedings of the 30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1616–1635, 2019.
9. Sepehr Assadi, Yu Chen, and Sanjeev Khanna. Sublinear Algorithms for (Δ+1) Vertex Coloring. In *Proceedings of the 30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 767–786, 2019.
10. Sepehr Assadi and Sanjeev Khanna. Tight Bounds on the Round Complexity of the Distributed Maximum Coverage Problem. In *Proceedings of the 29th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2412–2431, 2018.

11 Sepehr Assadi, Xiaorui Sun, and Omri Weinstein. Massively Parallel Algorithms for Finding Well-Connected Components in Sparse Graphs. In *Proceedings of the 38th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 461–470, 2019.

12 Bahman Bahmani, Ravi Kumar, and Sergei Vassilvitskii. Densest Subgraph in Streaming and MapReduce. *PVLDB*, 5(5):454–465, 2012.

13 Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. *J. ACM*, 64(6), 2017.

14 Soheil Behnezhad, Sebastian Brandt, Mahsa Derakhshan, Manuela Fischer, MohammadTaghi Hajiaghayi, Richard M. Karp, and Jara Uitto. Massively Parallel Computation of Matching and MIS in Sparse Graphs. In *Proceedings of the 38th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 481–490, 2019.

15 Soheil Behnezhad, Mahsa Derakhshan, and MohammadTaghi Hajiaghayi. Semi-MapReduce Meets Congested Clique. *CoRR*, abs/1802.10297, 2018.

16 Soheil Behnezhad, Laxman Dhulipala, Hossein Esfandiari, Jakub Lacki, and Vahab Mirrokni. Near-Optimal Massively Parallel Graph Connectivity. In *Proceedings of the 60th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, 2019. To appear.

17 Soheil Behnezhad, MohammadTaghi Hajiaghayi, and David G. Harris. Exponentially Faster Massively Parallel Maximal Matching. In *Proceedings of the 60th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, 2019. To appear.

18 Gianfranco Bilardi, Michele Scquizzato, and Francesco Silvestri. A Lower Bound Technique for Communication in BSP. *ACM Trans. Parallel Comput.*, 4(3):14:1–14:27, 2018.

19 Ashok K. Chandra, Larry J. Stockmeyer, and Uzi Vishkin. Constant Depth Reducibility. *SIAM J. Comput.*, 13(2):423–439, 1984.

20 Yi-Jun Chang, Manuela Fischer, Mohsen Ghaffari, Jara Uitto, and Yufan Zheng. The Complexity of $(\Delta+1)$ Coloring in Congested Clique, Massively Parallel Computation, and Centralized Local Computation. In *Proceedings of the 38th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 471–480, 2019.

21 Stephen A. Cook and Pierre McKenzie. Problems Complete for Deterministic Logarithmic Space. *J. Algorithms*, 8(3):385–394, 1987.

22 Artur Czumaj, Jakub Lacki, Aleksander Madry, Slobodan Mitrovic, Krzysztof Onak, and Piotr Sankowski. Round compression for parallel matching algorithms. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 471–484, 2018.

23 Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

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

25 Kousha Etessami. Counting Quantifiers, Successor Relations, and Logarithmic Space. *J. Comput. Syst. Sci.*, 54(3):400–411, 1997.

26 Benjamin Fish, Jeremy Kun, Ádám Dániel Lelkes, Lev Reyzin, and György Turán. On the Computational Complexity of MapReduce. In *Proceedings of the 29th International Symposium on Distributed Computing (DISC)*, pages 1–15, 2015.

27 Fabian Frei and Koichi Wada. Efficient Circuit Simulation in MapReduce. In *Proceedings of the 30th International Symposium on Algorithms and Computation (ISAAC)*, 2019. To appear.

28 Buddhima Gamlath, Sagar Kale, Slobodan Mitrovic, and Ola Svensson. Weighted Matchings via Unweighted Augmentations. In *Proceedings of the 38th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 491–500, 2019.

29 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 37th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 129–138, 2018.

30 Mohsen Ghaffari, Fabian Kuhn, and Jara Uitto. Conditional Hardness Results for Massively Parallel Computation from Distributed Lower Bounds. In *Proceedings of the 60th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, 2019. To appear.

31 Mohsen Ghaffari and Jara Uitto. Sparsifying Distributed Algorithms with Ramifications in Massively Parallel Computation and Centralized Local Computation. In *Proceedings of the 30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1636–1653, 2019.

32 Michael T. Goodrich. Communication-efficient parallel sorting. *SIAM J. Comput.*, 29(2):416–432, 1999.

33 Michael T. Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, Searching, and Simulation in the MapReduce Framework. In *Proceedings of the 22nd International Symposium on Algorithms and Computation (ISAAC)*, pages 374–383, 2011.

34 MohammadTaghi Hajiaghayi, Saeed Seddighin, and Xiaorui Sun. Massively Parallel Approximation Algorithms for Edit Distance and Longest Common Subsequence. In *Proceedings of the 30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1654–1672, 2019.

35 James W. Hegeman and Sriram V. Pemmaraju. Lessons from the Congested Clique applied to MapReduce. *Theor. Comput. Sci.*, 608:268–281, 2015.

36 Sungjin Im and Benjamin Moseley. A Conditional Lower Bound on Graph Connectivity in MapReduce. *CoRR*, abs/1904.08954, 2019.

37 Sungjin Im, Benjamin Moseley, and Xiaorui Sun. Efficient massively parallel methods for dynamic programming. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 798–811, 2017.

38 Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2007 EuroSys Conference*, pages 59–72, 2007.

39 Riko Jacob, Tobias Lieber, and Nodari Sitchinava. On the Complexity of List Ranking in the Parallel External Memory Model. In *Proceedings of the 39th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 384–395, 2014.

40 Howard J. Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for MapReduce. In *Proceedings of the 21st annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 938–948, 2010.

41 Hartmut Klauck, Danupon Nanongkai, Gopal Pandurangan, and Peter Robinson. Distributed Computation of Large-Scale Graph Problems. In *Proceedings of the 26th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 391–410, 2015.

42 Janne H. Korhonen and Jukka Suomela. Towards a Complexity Theory for the Congested Clique. In *Proceedings of the 30th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 163–172, 2018.

43 Ravi Kumar, Benjamin Moseley, Sergei Vassilvitskii, and Andrea Vattani. Fast Greedy Algorithms in MapReduce and Streaming. *ACM Trans. Parallel Comput.*, 2(3), 2015.

44 Jakub Lacki, Slobodan Mitrovic, Krzysztof Onak, and Piotr Sankowski. Walking Randomly, Massively, and Efficiently. *CoRR*, abs/1907.05391, 2019.

45 Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. Filtering: a method for solving graph problems in MapReduce. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 85–94, 2011.

46 Harry R. Lewis and Christos H. Papadimitriou. Symmetric Space-Bounded Computation. *Theor. Comput. Sci.*, 19:161–187, 1982.

47 Philip D. MacKenzie and Vijaya Ramachandran. Computational bounds for fundamental problems on general-purpose parallel models. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 152–163, 1998.

48 Krzysztof Onak. Personal communication, 2019.

49 Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. Fast Distributed Algorithms for Connectivity and MST in Large Graphs. *ACM Trans. Parallel Comput.*, 5(1), 2018.

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

51 Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

52 Andrea Pietracaprina, Geppino Pucci, Matteo Riondato, Francesco Silvestri, and Eli Upfal. Space-round tradeoffs for MapReduce computations. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS)*, pages 235–244, 2012.

53 Vibhor Rastogi, Ashwin Machanavajjhala, Laukik Chitnis, and Anish Das Sarma. Finding connected components in Map-Reduce in logarithmic rounds. In *Proceedings of the 29th IEEE International Conference on Data Engineering (ICDE)*, pages 50–61, 2013.

54 Omer Reingold. Undirected connectivity in log-space. *J. ACM*, 55(4):17:1–17:24, 2008.

55 Tim Roughgarden, Sergei Vassilvitskii, and Joshua R. Wang. Shuffles and Circuits (On Lower Bounds for Modern Parallel Computation). *J. ACM*, 65(6), 2018.

56 Michele Scquizzato and Francesco Silvestri. Communication Lower Bounds for Distributed-Memory Computations. In *Proceedings of the 31st International Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 627–638, 2014.

57 Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.

58 Leslie G. Valiant. A Bridging Model for Parallel Computation. *Commun. ACM*, 33(8):103–111, 1990.

59 Virginia Vassilevska Williams. On some fine-grained questions in algorithms and complexity. In *Proceedings of the International Congress of Mathematicians 2018 (ICM 2018)*, pages 3431–3472, 2018.

60 Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012.

61 Grigory Yaroslavtsev and Adithya Vadapalli. Massively parallel algorithms and hardness for single-linkage clustering under ℓ_p-distances. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*, pages 5596–5605, 2018.

62 Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache Spark: a unified engine for big data processing. *Commun. ACM*, 59(11):56–65, 2016.

APPENDIX

A L-Completeness of the One Cycle vs. Two Cycles Problem

In [21, Theorem 3] it is shown that graph connectivity when the given graph is known to be a disjoint union of cycles is L-hard. A careful inspection of the reductions used to establish this result reveals that the problem remains hard even when the graph is known to be made up of either one or three cycles. By reducing from a different problem, we show that it remains hard even when the graph is known to be made up of either one or two cycles.

▶ **Proposition 20.** *Graph connectivity for promise graphs that are either one cycle or two cycles is L-complete.*

Proof. Membership in L is guaranteed by the algorithm of Reingold [54]. To show L-hardness, we shall exhibit an NC^1 reduction from order between vertices. Given an instance (G, a, b) for order between vertices, we build a new graph G' as follows: (1) the two arcs pointing to a and to b, denoted (a', a) and (b', b), respectively, are removed, (2) the direction of each of the remaining $n - 3$ arcs is discarded, and (3) edges $\{s, a\}$, $\{a', b'\}$, and $\{b, t\}$ are added, where s denotes the source and t the sink of G, respectively. This construction is an NC^1 reduction. The resulting graph G' consists of two cycles if a precedes b in G, and of one single cycle otherwise. ◀

Sparse Hopsets in Congested Clique

Yasamin Nazari
Johns Hopkins University, Baltimore, MD, United States
ynazari@jhu.edu

Abstract

We give the first Congested Clique algorithm that computes a sparse hopset with polylogarithmic hopbound in polylogarithmic time. Given a graph $G = (V, E)$, a (β, ϵ)-hopset H with "hopbound" β, is a set of edges added to G such that for any pair of nodes u and v in G there is a path with at most β hops in $G \cup H$ with length within $(1 + \epsilon)$ of the shortest path between u and v in G. Our hopsets are significantly sparser than the recent construction of Censor-Hillel et al. [6], that constructs a hopset of size $\tilde{O}(n^{3/2})$[1], but with a smaller polylogarithmic hopbound. On the other hand, the previously known construction of sparse hopsets with polylogarithmic hopbound in the Congested Clique model, proposed by Elkin and Neiman [10, 11, 12], all require polynomial rounds.

One tool that we use is an efficient algorithm that constructs an ℓ-limited neighborhood cover, that may be of independent interest. Finally, as a side result, we also give a hopset construction in a *variant* of the low-memory Massively Parallel Computation model, with improved running time over existing algorithms.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Hopsets, Congested Clique, Shortest Paths, Massively Parallel Computation

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.34

Funding This work supported is in part by NSF awards CCF-1464239 and CCF-1909111.

Acknowledgements The author would like to thank Michael Dinitz and Goran Zuzic for helpful discussions.

1 Introduction

In the Congested Clique model of distributed computing, we are given a graph with n nodes, where all nodes can send a (possibly different) message with $O(\log(n))$-bits to *every* other node in the graph in each round. In the context of distributed graph algorithms, the input graph is a subset of the communication graph. In addition to theoretical interest in this model, it has also recently gained a lot of attention, due to its connections to practical distributed and big data platforms such as MapReduce (e.g. [14]) and related platforms such as Spark and Hadoop (e.g. [5]).

Distance problems, such as single-source shortest path (SSSP) and multi-source shortest path (MSSP), have been widely studied in different models. A fundamental structure that has been used for solving these problems is a hopset. Given a graph $G = (V, E)$, a (β, ϵ)-hopset H with hopbound β, is a set of edges added to G such that for any pair of nodes u and v in G, there is a path with at most β hops in $G \cup H$ with length within $(1 + \epsilon)$ of the shortest path between u and v in G. The approximation ratio is also referred to as distortion or *stretch*. We generally want to have sparse hopsets with small hopbound. Intuitively, a hopset can be seen as adding a number of "shortcut edges" that serve as reducing the graph diameter at the expense of a small loss of accuracy. Once a hopset is preprocessed, we can use it as many times as needed for distance queries, and the query time will be the hopbound.

[1] The $\tilde{O}(f(n))$ notation is used to hide $O(\log(f(n)))$ factors.

© Yasamin Nazari;
licensed under Creative Commons License CC-BY
23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller; Article No. 34; pp. 34:1–34:16
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

There is a natural tradeoff between the size and the hop-bound (or the query time) of a hopset. In an extreme-case one could store the complete adjacency list-or equivalently add $O(n^2)$ edges, and then query distance in constant time. Other than the fact that computing all-pairs shortest-path is generally slow, we often do not have enough space to store the whole adjacency list for large-scale graphs. There is a line of work that focuses on designing data structures with small size, say $\tilde{O}(n^{1+1/k})$, in which distances can be estimated up to $O(k)$ stretch in small query time. Examples of such structures are Thorup-Zwick distance oracles [24] or k-spanners. Hopsets offer a different tradeoff: a hopset gives an accuracy of $1+\epsilon$ (rather than $O(k)$) at the expense of a larger query time (polylogarithmic instead of a small constant). It is therefore crucial to keep the hopbound as small as possible, since the hopbound will basically determine the query time and is more important than preprocessing time. However, even in centralized settings there are existential limitations in this tradeoff. There is a lower bound argument by [1] stating that there are graphs for which we can not have a hopbound of $o(\log(k)/\epsilon)^{\log(k)}$ and size $O(n^{1+1/k})$, for arbitrary $0 < \epsilon < 1$.

In a recent result, Censor-hillel et al. [6] gave a fast Congested Clique algorithm that constructs a hopset with hopbound $O(\log^2(n)/\epsilon)$ and size $\tilde{O}(n^{3/2})$. While we can use their hopsets to compute distances efficiently, one shortcoming of such a construction is the large space. In particular, if the original graph has size $o(n^{3/2})$, we would be storing more edges than the initial input. This is undesirable due to the large scale nature of data in modern distributed platforms. It is therefore natural to find algorithms that use less space, possibly in exchange for a slightly weaker hopbound (but still polylogarithmic). This is our main goal in this paper. We extend the result of [6] by constructing sparse hopsets with size $\tilde{O}(n^{1+1/k})$ for a constant $k \geq 2$ and polylogarithmic hopbound in polylogarithmic time in Congested Clique. This is the first Congested Clique construction of sparse hopsets with polylogarithmic hopbound that uses only *polylogarithmic* number of rounds. This implies that we can store a sparse auxiliary data structure that can be used later to query distances (from multiple sources) in polylogarithmic time.

Our hopset construction is based on a combination of techniques used in Cohen [8] (with some modifications) and the centralized construction of Huang and Pettie [16]. We also use another result of [6] that allows us to efficiently compute $(1+\epsilon)$-approximate multi-source shortest path distances from $O(\sqrt{n})$ sources.

One tool that we use in our construction is a hop-limited neighborhood cover construction, which may be of independent interest. Roughly speaking, a W-neighborhood cover is a collection of clusters, such that there is a cluster that contains the neighborhood of radius W around each node, and such that each node overlaps with at most $O(\log(n))$ clusters. In an ℓ-limited W-neighborhood cover only balls with radius W using paths with at most ℓ-hops are contained in a cluster.

We note that many of the techniques we use in our construction are borrowed from the PRAM literature. We hope that this paper provides some insight into connections between these different but relevant models.

1.1 Our contribution

Congested Clique. The state-of-the-art construction of sparse hopsets in Congested Clique is the results of Elkin and Neiman [12] (and similar results in [10, 11]), but these algorithms requires polynomial number of rounds for constructing hopsets with polylogarithmic hopbound. The construction of Censor-Hillel et al. [6] is a special case of hopsets of [10]. They construct hopsets of size $\tilde{O}(n^{3/2})$ with $O(\log^2(n)/\epsilon)$ hopbounds. They can construct such a hopset in $O(\log^2(n)/\epsilon)$ rounds of Congested Clique using sparse matrix multiplication techniques.

However, [6] does not give any explicit results for constructing sparser hopsets. It is possible that their techniques will also lead to faster Congested Clique algorithms for constructing general (sparse) hopsets of [10, 12]. But here we use a new hopset construction that has a very different structure than hopsets of [10] and with improved guarantees. Not only does our hopset construction run in polylogarithmic rounds, but it also yields a better size and hopbound tradeoff over the state-of-the-art Congested Clique construction of [12]. Prior to [11] and [6] the hopsets proposed for Congested Clique had superpolylogarithmic hopbound of $2^{\tilde{O}(\sqrt{\log(n)})}$ [15] or polynomial [22] hopbound. More formally, we provide an algorithm with the following guarantees:

▶ **Theorem 1.** *Given a weighted*[2] *graph $G = (V, E, w)$, for any $k \geq 2, 0 < \epsilon \leq 1$, there is a Congested Clique algorithm that computes a (β, ϵ)-hopset of size $O(n^{1+\frac{1}{2k}} \log(n) + n \log^2(n))$ with hopbound $\beta = O(\frac{\log^2(n)}{\epsilon}(\frac{\log(n)\log(k)}{\epsilon})^{\log(k+1)-1})$ with high probability in $O(\beta \log^2(n))$ rounds.*

To comapre this with the *efficient variant* of the Congested Clique hopsets of [11], we note that for a hopset of size $O(n^{1+\frac{1}{k}} \log(n))$, for a constant k, we get a hopbound $O(\frac{\log^2(n)}{\epsilon}(\frac{\log(n)\log(k)}{\epsilon})^{\log(k+1)-2})$, whereas [11] gets hopbound of $\Omega((\frac{\log(n)\log(k)}{\epsilon})^{\log(k)+2})$. Thus our hopbound guarantee is a factor of $\Theta(\log^{2-o(1)}(n)(\frac{\log(k)}{\epsilon})^{4-o(1)})$ improvement over construction [11]. Also, their more efficient algorithm runs in $\tilde{O}(n^{\rho})$ rounds of Congested Clique, where $0 < \rho \leq \frac{1}{2}$ is a parameter that impacts the hopbound (ρ is a constant when their hopbound is polylogarithmic). They have another algorithm that uses an extra polynomial factor in running time to obtain constant hopbound[3]. We note that the construction of [12] has similar guarantees to [11], with two differences: it eliminates a $\log(n)$ factor (or more generally the dependence on aspect ratio) in the hopset size, but has a slightly worse hopbound in their fastest regime.

Our construction is mainly based on the ideas of [8] with a few key differences that take advantage of the power of Congested Clique. While the hopsets of [11] significantly improve over hopsets of [8] in centralized settings, the construction of [8] has certain properties that makes it adaptable for a better Congested Clique algorithm. In particular, [8] uses a notion of small and big clusters, and we can utilize this separation in Congested Clique. We change the algorithm of [8], in such a way that leads to adding fewer edges for small clusters. This leads to sparser hopsets and improves the overall round complexity. The key idea is that by using the right parameter settings, in Congested Clique we can send the whole topology of a small cluster to a single node, *the cluster center*, and then compute the best known hopset locally. It is possible to perform these operations specifically in Congested Clique due to a well-known routing algorithm by Lenzen [18]. We can then combine Theorem 1 with a *source detection* algorithm by [6] (formally stated in Lemma 6) to get the following result for computing multiple-source shortest path queries.

▶ **Corollary 2.** *Given a weighted graph $G = (V, E, w)$ there is a Congested Clique algorithm that constructs a data structure of size $O(m + n^{1+\frac{1}{2k}})$ in $O(\beta \log^2(n))$ rounds, where $\beta = O(\frac{\log^2(n)}{\epsilon}(\frac{\log(n)\log(k)}{\epsilon})^{\log(k+1)-1})$, such that after construction we can query $(1+\epsilon)$-stretch distances from $O(\sqrt{n})$ sources to all nodes in V in $O(\beta)$ rounds with high probability.*

---

[2] For simplicity, we assume that the weights are polynomial. This assumption can be relaxed using standard reductions that introduce extra polylogarithmic factors in time (and hopbound) (e.g. see [17],[20], [11]).

[3] If we allow extra polynomial factors in the running time we may also get a constant hopbound (we would need to change the parameters of the neighborhood cover construction, and change how we iteratively use smaller scales). However this is inconsistent with our main motivation of getting polylogarithmic round complexity.

In a related result, the problem of single-source shortest path (SSSP) in Congested Clique was also studied in [4], where they use continuous optimization techniques for solving transshipment. Firstly, their algorithm takes a large polylogarithmic round complexity, and has a high dependence on ϵ. But we can have a significantly smaller running time depending on the hopset size. In other words, for hopsets with a reasonable density (e.g. with size $n^{1+\mu}$, where $\mu < 0.1$) we get a much smaller polylogarithmic factor for computing $(1+\epsilon)$-SSSP. This can be further reduced if we allow denser hopsets.

More importantly, an approach such as [4] is mainly suited for SSSP. One limitation with their approach is that for computing multiple distance queries we need to repeat the algorithm for each query. For example, for computing the shortest path from s sources to all nodes, we have to repeat the whole algorithm s times. But constructing a hopset will let us run many such queries in parallel in $O(\beta + s)$ rounds, where β is the hopbound. Moreover, we can compute multi-source shortest path from $O(\sqrt{n})$ sources in parallel for all the sources using the source detection algorithm of [6].

Neighborhood and Pairwise Covers. In Section 4, we focus on an efficient construction of a *limited* pairwise cover (or neighborhood cover) in the CONGEST model, which is a tool that we use in our hopset construction. Given a weighted graph $G = (V, E)$, a W-pairwise cover, as defined by [7], is a collection $\mathcal{C}$ of subsets of V with the following properties. 1) the diameter of each cluster is $O(W \log n)$, 2) $\sum_{C \in \mathcal{C}} |C| = O(n)$, $\sum_{C \in \mathcal{C}} E(C) = \tilde{O}(m)$. In other words, the sum of the sizes of all clusters is $O(n)$, and the sum of all edge occurrences in the clusters is $\tilde{O}(m)$, 3) for every path p with (weighted) length at most W, there exists a cluster C where $p \subseteq C$.

Pairwise covers are similar to neighborhood covers of Awerbuch and Peleg [2] with two differences: in a W-neighborhood cover, there must be a cluster that contains the neighborhood of radius W around each node rather than only paths of length W. Neighborhood covers also need an additional property that *each node* is in at most $O(\log(n))$ clusters. While for our purposes the path covering property is enough, in distributed settings we need the property that each node overlaps with few clusters to ensure that there is no congestion bottleneck. The main subtlety in constructing a general W-pairwise (or neighborhood) cover is that we may need to explore paths of $\Omega(n)$ hops, and thus it is not clear how this can be done in polylogarithmic time. To resolve this, [7] proposed a relaxed construction called *ℓ-limited W-pairwise cover*. This structure has all the above properties but only for paths with at most ℓ-hops. More formally, the third property will be relaxed to require that for every path p of weight at most W with at most ℓ hops there exists a cluster C where $p \subseteq C$. We can define an ℓ-limited W-neighborhood cover similarly.

A randomized algorithm that constructs ℓ-limited pairwise covers with high probability in $O(\ell)$ depth in the PRAM model was given by [7]. The ideas used in [20] for constructing work-efficient PRAM hopsets, can also be used to construct ℓ-limited pairwise and neighborhood covers in PRAM. However, they do not explicitly construct limited pairwise covers. In distributed settings, a recent construction for *sparse* neighborhood covers in *unweighted graphs* in the CONGEST model was given by [23]. However, even by generalizing their result to weighted graphs, in order to cover distances for large values of W the algorithm would take $\tilde{\Omega}(W)$ rounds for reasons described above.

To the best of our knowledge, an efficient algorithm for constructing ℓ-limited pairwise-covers (or limited neighborhood covers) is not directly studied in the CONGEST and Congested Clique literature. Our first contribution is such an algorithm: we use the low-diameter decomposition construction of Miller et al. [20] for *weighted graphs*, combined with

a rounding technique due to [17] to construct ℓ-limited W-pairwise covers in $O(\ell \log^2(n))$ rounds in the CONGEST model. Importantly, ℓ is a parameter independent of W, which we will set to a polylogarithmic value throughout our hopset construction. Our algorithm is similar to the algorithm of [20], but with some adaptations needed for implementation in the CONGEST model. Formally, we get the following result:

▶ **Theorem 3.** *Given a weighted graph $G = (V, E, w)$, there is an algorithm that constructs an ℓ-limited W-pairwise cover in $O(\ell \log^2(n) \log(W))$ rounds in the CONGEST model, with high probability. Moreover, a pairwise cover for paths with ℓ-hops with length in $[W, 2W]$* [4] *can be constructed in $O(\ell \log^2(n))$ rounds with high probability.*

MPC. As a side result[5], we note that pairwise covers can also be constructed efficiently in the Massively Parallel Computation model (MPC) (even when memory per machine is strictly sublinear). This in turn leads to a better running time for $(1+\epsilon)$-MSSP from $O(\sqrt{n})$ sources (and consequently SSSP), in $O(\log^2(n)/\epsilon)$ rounds in a variant of the model where we assume the overall memory of $\tilde{O}(m\sqrt{n})$ (equivalently, we have *more machines* than in the standard MPC model). We consider this variant since in practice it is plausible that there are more machines, while due to the large-scale nature of data in these settings, using less memory per machine is often more crucial.

Dinitz and Nazari [9] construct (β, ϵ)-hopsets (based on hopsets of [11]) with polylogarithmic hopbound when the overall memory is $\tilde{O}(m)$, but they argue that using the existing hopset constructions, this would take polynomial number of rounds in MPC. They further show that if the overall memory is by a polynomial factor larger (i.e. if the overall memory is $\tilde{\Theta}(mn^\rho)$ for a constant $0 < \rho \leq 1/2$), then hopsets with polylogarithmic hopbound can be constructed in polylogarithmic time. We can also use this extra-memory idea, first to argue that using the hopsets of [8] instead of the hopsets of [11] we can get a smaller hopbound when the overall memory is $O(m\sqrt{n})$. Then we observe that if we use a faster ℓ-limited pairwise cover algorithm (based on the construction of [20]) instead of the pairwise covers that [8] uses, we can shave off a polylogarithmic factor in the construction time. This ℓ-limited W-pairwise cover construction may also be of independent interest in MPC. More formally, we get faster algorithms for $(1+\epsilon)$-MSSP:

▶ **Theorem 4.** *Given an undirected weighted graph G, we can compute $(1+\epsilon)$-MSSP from $O(n^{1/2})$ sources in $O(\frac{\log^2(n)}{\gamma\epsilon})$ rounds of MPC, when memory per machine is $\tilde{O}(n^\gamma), 0 < \gamma \leq 1$ and the overall memory is $\tilde{O}(mn^{1/2})$ (i.e. there are $\Theta(mn^{1/2-\gamma})$ machines).*

The difference between this result and [9] is that they give a more general result where the overall memory is $\tilde{O}(mn^\rho)$ for a parameter $\rho > 0$. But in the special case of $\rho = 1/2$, we get a hopbound of $O(\frac{\log^2(n)}{\epsilon})$, whereas in this case they get a hopbound of $O((\frac{\log(n)}{\epsilon})^3)$. We also note that the main focus of [9] is constructing Thorup-Zwick distance sketches. As explained earlier, these structures offer a different tradeoff: much weaker accuracy ($O(k)$-stretch), but better query time (constant rounds rather than polylogarithmic) and less space after preprocessing ($\tilde{O}(n^{1+1/k})$ instead $\tilde{O}(m)$ in the case of hopsets). More details on the MPC algorithm can be found in Section 6.

[4] The algorithm and analysis can easily be extended to paths with length $[W, cW]$ for any constant c.

[5] Our MPC results can be seen as a straight-forward combination of results of [8], [20], [9] and simulation of [13]. But since both the construction and the model are closely relevant to our Congested Clique algorithms, we find it useful to include this discussion.

1.2 Overview of techniques

Our hopset has a similar structure to hopsets of [8], but with some changes both in construction and the analysis. We also take advantage of multiple primitives that are specific to Congested Clique such as Lenzen's routing and a recent result of [6]. First, we explain the ℓ-limited W-neighborhood cover construction and then we explain the hopset algorithm.

ℓ-limited neighborhood covers. As described earlier, our algorithm for constructing a W-neighborhood cover is based on a combination of the low-diameter decomposition of [20], and a rounding technique originally proposed by [17]. At a high level, in the low-diameter decomposition algorithm of [21, 20], each node u chooses a radius r_u based on an exponential random variable. Then each node u joins the cluster of node v that minimizes *the shifted distance* from u, which is defined as $d(u, v) - r_v$. This leads to a partition of the graph, and we can show that by repeating this process we will get a W-neighborhood cover. Since partitions for constructing a W-neighborhood covers directly will be slow for large values of W, we focus on the ℓ-limited W-pairwise covers. To construct these, consider all pairs of nodes within distance $[w, 2w], w \leq W$ in each iteration. We round up the weights of each edge in the graph based on values w and ℓ. We then construct a low-diameter-decomposition *based on the new weights*, such that the diameter of each cluster is $O(\ell \log(n))$ (rather than $O(W \log(n))$ based on the original weights). The rounding scheme is such that the ℓ-limited paths with length $[w, 2w]$ in the original graph will be explored. Intuitively, this means that on the rounded graph we need to explore a neighborhood with fewer hops, which will lead to a faster construction. We can then repeat this process for $O(\log(W))$ times for different distance intervals. The details of this rounding scheme can be found in Section 4.

Hopset Construction. First we describe the sequential hopset construction and will then choose the parameters appropriately for our distributed construction. Let μ be a a parameter that we will set later. The (sequential) structure of the hopset is as follows: In each iteration we consider pair of nodes $u, v \in V$ such that $R \leq d(u, v) < 2R$, and we call the interval $[R, 2R)$ a *distance scale*. Then for distance scales $[R, 2R)$ we set $W = O(\epsilon R/(\log n))$ and construct a W-pairwise cover. We let big clusters be the clusters that have size at least n^μ and small clusters have size less than n^μ, where $0 < \mu < 1$ is a constant parameter. Then we construct a hopset with small hopbound on each of the small clusters. This is the main structural difference with the construction of [8] that adds a clique for the smallest hopsets. We then add a star from the center node of each big cluster to every other node in that cluster, and add a complete graph at the center of large clusters. Whenever we add an edge, we set the weight to be the distance between the two endpoints. In the distributed construction, the weight will be an estimate of this distance that we will describe later.

Roughly speaking, constructing a hopset on small clusters rather than constructing a clique as [8] does, allows us to set the size threshold of small clusters larger, while keeping the number of edges added small. This in turn reduces the number of big cluster centers we have to deal with. Such a modification can be very well tuned to the Congested Clique model. By setting $\mu = 1/2$, we will have small clusters that will have at most $O(n)$ edges. Then a well-known routing algorithm by Lenzen [18] can be used to send all these edges to the cluster center. The cluster center can then compute a hopset locally. For this we use current best-known centralized construction by [16]. The other challenge is that we need to compute pairwise distances between all big cluster centers. In [8] this step is done by running Bellman-Ford instances from different sources in parallel. But directly implementing this in distributed settings would need $\Omega(\sqrt{n})$ rounds due to congestion. This is where we use a

recent result by [6] stating that we can compute $(1+\epsilon)$-approximate distances from $O(\sqrt{n})$ sources in $O(\log^2(n)/\epsilon)$ time. We point out that in [8], in order to get sparse hopsets, they use a recursive construction for small clusters. Such a recursion would introduce significant overhead in the hopbound guarantee. Here we show that in Congested Clique by using the tools described above we can avoid using the recursive construction and still compute sparse hopsets.

We explain briefly why the constructed hopset has the size and hopbound properties stated in Theorem 1. To see this, we use similar arguments as in [8]: for a distance scale $[R, 2R)$ consider a shortest path of length at most $2R$, and consider $O(\log(n)/\epsilon)$ segments of length W on this path. By definition of a W-pairwise cover, each such segment is contained in a cluster. If this segment is in a small cluster, there is a corresponding path with at most β' edges, where β' is the hopbound of the local construction. For big clusters, we either add a single edge, or if there is more than one big cluster, the whole segment between these clusters has a corresponding edge in the hopset. By similar considerations and by the triangle inequality we can show that the stretch of the replaced path is $(1+\epsilon)$. We need a tighter size analysis than the one used in [8] to prove the desired sparsity. We use a straight-forward bucketing argument as follows: for each cluster of size $\Theta(s)$, $\Theta(s^{1+1/k})$ edges will be added. Then by noting that are at most $O(n/s)$ clusters with this size we can bound the overall size.

Bounding the exploration depth. For large values of R, the shortest path explorations up to distance R could take $\Omega(n)$ rounds in distributed settings. To keep the round complexity small, we use the following idea from [8] (also used in [11] and [6]): we can use the hopset edges constructed for smaller distance scales for constructing hopset edges for larger distance scales more efficiently. The intuition behind this idea is that any path with length $[R, 2R)$ can be divided into two segments, such that for each of these segments we already have a $(1+\epsilon)$-stretch path with β hops using the edges added for smaller distance scales. This allows us to limit the explorations only to paths with $2\beta+1$ hops in each iteration. This process will impact the accuracy, and so in order to keep the error small we have to construct the hopsets for a fixed scale at a higher accuracy. This is where a factor polylogarithmic in n will be introduced in the hopbound, which can generally be avoided in the centralized constructions (e.g. see [8, 11]). This idea is formalized in Lemma 5.

2 Models and Notation

Given a weighted undirected graph $G = (V, E, w)$, and a pair $u, v \in V$ we denote the (weighted) shortest path distance by $d(u, v)$. We denote by $d^\ell(u, v)$ the length of the shortest path between u and v among the paths that use at most ℓ hops, and call this the ℓ-hop limited distance between u and v. For each node $v \in V$, we denote the (weighted) radius r neighborhood around v by $B(v, r)$, and we let $B^\ell(v, r)$ be the set of all nodes $u \in V$ such that there is path π of (weighted) length at most r between u and v such that π has at most ℓ hops.

For parameter $\beta > 0, \epsilon > 0$, a graph $G_H = (V, H, w_H)$ is called a (β, ϵ)-hopset for the graph G, if in graph $G' = (V, E \cup H, w')$ obtained by adding edges of G_H, we have $d_G(u, v) \leq d^\beta_{G'}(u, v) \leq (1+\epsilon) d_G(u, v)$ for every pair $u, v \in V$ of vertices. The parameter β is called the *hopbound* of the hopset.

Models. We construct limited neighborhood covers in the more classical *CONGEST* model, in which we are given an undirected graph $G = (V, E)$, and in each round nodes can send

a message of $O(\log(n))$-bits to each of their neighbors in G (different messages can be sent along different edges). In the *Congested Clique* model, we are given a graph with n nodes, where all nodes can send a message with $O(\log(n))$-bits to *every* other node in the graph in each round [19]. In other words, this is a stronger variant of the CONGEST model, in all nodes can communicate with each other directly.

We also consider the *Massively Parallel Computation*, or MPC model [3]. In this model, an input of size N which is arbitrarily distributed over N/S machines, each of which has $S = N^\gamma$ memory for some $0 < \gamma < 1$. In the standard MPC model, every machine can communicate with every other to at most S other machines arbitrarily. Generally, for graph problems the total memory N is $O(m), m = |E|$ words. But here we a consider a variation of the model in which the total memory can be larger, while the memory per machine is strictly sublinear in n. In other words, each machine has $O(n^\gamma), \gamma < 1$ memory, where $n = |V|$.

Even though we do not give any new PRAM results, we use multiple tools from PRAM literature. In the PRAM model[6], a set of processors perform computations by reading and writing on a shared memory in parallel. The total amount of computation performed by all processors is called the *work*, and the number of parallel rounds is called the *depth*.

3 Algorithmic Tools.

In this section we describe several algorithmic tools from previous work that we will be using.

Bounding the shortest path exploration. As explained earlier, for an efficient hopset construction, we need to first compute hopsets for smaller distance scales and then use the new edges for computing future distances. This will let us limit the shortest path explorations to a logarithmic number of hops in each round. More formally,

▶ **Lemma 5** ([8, 11]). *Let H^k be the hopset edges for distance scale $[2^{k-1}, 2^k)$ with hopbound β. Then for any pair u, v where $d(u, v) \in [2^k, 2^{k+1})$, there is a path with $2\beta + 1$ hops in $G \cup (\cup_{i=\log(\beta)}^{k} H^i)$ with length $(1 + \epsilon)$-approximate of the shortest path between u and v.*

Roughly speaking, the above lemma implies that we can use previously added edges and only run Bellman-Ford for $2\beta + 1$ rounds for each iteration of our algorithm.

Lenzen's routing. Given a set of messages such that each node is source and destination of at most $O(n)$ messages, these messages can all be routed to their destination in $O(1)$ time in Congested Clique [18].

Multi-source shortest path and source detection in Congested Clique. We use the following two results by [6]. First result is a multi-source shortest path algorithm that we use as a subroutine in our hopset construction:

▶ **Lemma 6** (MSSP, [6]). *Given a weighted and undirected graph, there is an algorithm that computes $(1 + \epsilon)$-approximate distances distances from a set of $O(\sqrt{n})$ sources in $O(\frac{\log^2(n)}{\epsilon})$ rounds in the Congested Clique model.*

The second result solves a special case of the so-called *source-detection* problem that we use to prove Corollary 2:

[6] We just use a simple abstraction without details of the exact parallel model (EREW, CRCW, etc), since PRAM is not our focus and there are reductions with small overhead between these variants.

▶ **Lemma 7** (Source detection, [6]). *Given a fixed set of $O(\sqrt{n})$ sources S, we can compute ℓ-hop limited distances from all nodes to each of the nodes in S in $O(\ell)$ rounds in the Congested Clique model.*

4 Neighborhood covers using low-diameter decomposition

In this section, we describe an algorithm for constructing pairwise covers in the CONGEST model. We first give an algorithm for W-pairwise covers in weighted graphs that runs in $O(W \log^2(n))$ rounds. We then provide an ℓ-limited W-pairwise cover that runs in $O(\ell \log^2(n))$ rounds. Clearly, the CONGEST algorithm can also be used in Congested Clique with the same guarantees. We will use the low-diameter decomposition algorithm that was proposed in [21] and extended (to weighted graphs) in [20] for computing pairwise covers in PRAM. First we state their PRAM result:

▶ **Theorem 8** (MPX [20, 21]). *Given a weighted an undirected graph $G = (V, E, w)$, there is a randomized parallel algorithm that partitions V into clusters $\mathcal{X}_1, \mathcal{X}_2, ...$ such that w.h.p. the (strong) diameter of each cluster $\mathcal{X}_i$ is at most $O(\frac{\log(n)}{\alpha})$. This algorithm has $O(\alpha^{-1} \log(n))$ depth[7] w.h.p. and $O(m)$ work.*

We denote the algorithm of [20] for a parameter $0 < \alpha < 1$ by LDD(α), which is as follows: each node $u \in V$ first chooses a random radius r_u based on an exponential distribution $\exp(\alpha)$. Each node $v \in V$ joins the cluster of node $u = \arg\min_{x \in V}(d(v, x) - d_x)$. Ties can be broken aribtrarily. It is easy to see that based on simple properties of exponential random variables the weak diameter of each cluster is $O(\alpha^{-1} \log(n))$ with high probability. But it can be shown that the clusters also have strong bounded diameter of $O(\alpha^{-1} \log(n))$ (as argued in [21, 20]). This means the diameter of the subgraph induced by each cluster is $O(\alpha^{-1} \log(n))$ as opposed to the weak diameter guarantee, which bounds the diameter with between each pair of nodes in the cluster based on distaces in G. The second property is that we can lower bound the probability that the neighborhood around each node is fully contained inside one cluster by a constant. This was shown in [20], but we give a proof sketch for completeness.

▶ **Lemma 9** (Padding property[8], [20]). *Let $\mathcal{X}$ be a partition in support of the LDD(α) algorithm. For each node $u \in V$, the probability that there exists $C \in \mathcal{X}$ such that $B(u, r) \subseteq C$ is at least $\exp(-2r\alpha)$.*

Proof sketch. For each node u we will consider the subgraph induced by $B(u, r)$. For each node $v \in V$, consider the random variable $Y_v := r_v - d(u, v)$. Let Y_1 denote the largest Y_v over $v \in B(u, r)$, and let Y_2 denote the second largest value. We argue that the probability that $B(u, r)$ intersects more than one cluster is at most $1 - \exp(-2r\alpha)$. This event occurs when Y_1 and Y_2 are within $2r$ of each other. Therefore we only need to bound the probability that $Y_1 - Y_2 < 2r$. This now follows from Lemma 4.4. of [21] that claims the following: given a sequence of exponential random variables $r_1, r_2, ..., r_n$, and arbitrary values $d_1, d_2, ..., d_n$ the probability that largest and second largest values $r_i - d_i$ are within δ of each other is at most $1 - \exp(-\delta\alpha)$. This implies the probability that $B(u, r)$ intersects more than one cluster is at most $1 - \exp(-2r\alpha)$ and this proves the claim. For more details see [21], [20]. ◀

[7] Depending on the exact PRAM model considered the depth may have a small extra factor of $O(\log^*(n))$.
[8] Lemma 2.2 in [20] upper bounds the probability that a ball overlaps with k or more clusters, but Lemma 9 is a straightforward corollary of this claim.

In order to compute W-neighborhood covers *sequentially*, we can use the above theorem by setting $\alpha = 1/W$ and repeating the partition algorithm $O(\log(n))$ times. It follows from a standard Chernoff bound that the desired properties hold with high probability. Implementing this algorithm in distributed settings may take $\Omega(n)$ rounds for large values of W. To resolve this, we use a relaxed notion similar to the notion of *ℓ-limited W-pairwise cover* proposed in [8]. This structure has all the properties of a W-pairwise cover but the path covering property only holds for paths with at most ℓ-hops. More formally, the third property will be relaxed to require that for every path p of weight at most W with at most ℓ hops there exists a cluster C where $p \subseteq C$. We define an ℓ-limited W-neighborhood cover similarly: for each node u, there is a cluster C such that $B^\ell(u, W) \subseteq C$. In [8], Cohen shows that we can construct ℓ-limited W-pairwise covers in $O(\ell)$ parallel depth, independent of W. We will show that this concept can also be utilized to limit the number of rounds for LDD(α) partitions to $O(\ell)$.

ℓ-limited W-pairwise cover. Since running LDD(α) by setting $\alpha = 1/W$ will require many rounds, we cannot directly use the weighted variant of LDD(α). Instead we use a rounding idea that allows us to run LDD(α) on the graph obtained from rounded weights, only for $\alpha = O(1/\ell)$, at the cost of a small loss in accuracy. This idea was proposed by [17] and is used widely in PRAM literature (e.g. [8], [20]). In the context of distributed algorithms a similar approach was used by [22] in CONGEST, but directly applying the result of [22] to our settings will require a polynomial running time, since we would need to run the algorithm from many (polynomial) sources. The idea is based on the following observation: consider a path π with at most ℓ hops, such that $R \leq w(\pi) \leq 2R$ for a fixed $R > 0$. Then by slightly changing the weights of each edge $e \in \pi$ by a small additive factor such that for the new weight $\hat{w}$ it holds $w(e) \leq \hat{w}(e) \leq w(e) + \frac{\epsilon_0 R}{\ell}$ for an arbitrary $\epsilon_0 > 0$. We then get $\hat{w}(\pi) \leq w(\pi) + R\epsilon_0 \leq (1 + \epsilon_0)w(\pi)$. This can be achieved by setting $\hat{w}(e) = \lceil \frac{w(e)}{\eta} \rceil$, where $\eta = \frac{\epsilon_0 R}{\ell}$.

▶ **Lemma 10** ([17]). *Given a weighted graph $G = (V, E, w)$, and a parameter R, there is a rounding scheme that constructs another graph $\hat{G} = G = (V, E, \hat{w})$ such that any path π with at most ℓ hops and weight $R \leq w(\pi) \leq 2R$ in G, has $\hat{w}(\pi) \leq \lceil 2\ell/\epsilon_0 \rceil$ in $\hat{G}$. Moreover, $w(\pi) \leq \eta(R, \ell) \cdot \hat{w}(\pi) \leq (1 + \epsilon)w(\pi)$, where $\eta(R, \ell) = \epsilon_0 R/\ell$.*

We can now run LDD(α) for $\alpha = O(\ell)$ on $\hat{G}$, and each path π with at most ℓ hops will be fully contained in some cluster with probability at least $\exp(-\ell \cdot O(1/\ell)) = \Omega(1)$. We can then recover an estimate to the original length $w(\pi)$ by setting $\tilde{w}(\pi) = \eta(R, \ell) \cdot \hat{w}(\pi)$, and we have $w(\pi) \leq \tilde{w}(\pi) \leq (1 + \epsilon_0) \leq w(\pi)$. Same as before, by repeating the LDD(α) algorithm $O(\log(n))$ times we will get an ℓ-limited W-neighborhood cover. We first argue that this algorithm can be implemented $O(\alpha \log^2(n))$ rounds of the CONGEST model. A similar construction was used in [23] for W-neighborhood covers in CONGEST. But the result of [23] only focuses on unweighted graphs, and would take $O(W \log(n))$ rounds.

▶ **Theorem 11.** *Given a weighted graph $G = (V, E, w)$, there is an algorithm that constructs an ℓ-limited W-pairwise cover in $O(\ell \log^2(n) \log(W))$ rounds in the CONGEST model, with high probability. Moreover, a pairwise cover for paths with ℓ-hops with length in $[W, 2W]$ can be constructed in $O(\ell \log^2(n))$ rounds with high probability.*

Proof. As we argued by using the rounding technique of [17], for any pair of nodes u, v such that $d^\ell(u, v) \in [W, 2W)$ we can restrict our attention to another graph $\hat{G}$ with rounded weights. We construct a pairwise cover on $\hat{G}$ by running the LDD(α), $\alpha = \epsilon_0/2\ell = \Theta(1/\ell)$ algorithm $O(\log(n))$ times independently.

We argue that each run of LDD($O(1/\ell)$) takes $O(\ell)$ rounds in the CONGEST model. First we observe that each node u only needs to broadcasts the value r_u to all the nodes within its r_u neighborhood, since a node x will not join the cluster of u if $r_u - d(u,x) < 0$. We can now use a simple induction to prove the claim. In each round, each node u will forward the radius and distances corresponding to the node $u_{\max}$ that maximizes $r_{u_{\max}} - d(u, u_{\max})$ among over all the messages that u has received. We now argue that each node u will receive the message from the node $c = \arg\max_{v \in V} r_v - d(u,v)$ in r_c rounds. Consider any path $\pi = \{c = u_0, u_1, ..., u_j = u\}$, where $j \leq r_c$. If $j = 1$ then in one single round c sends $(r_c, w(c, u_1))$ to u_1. Assume now that u_i receives the message $(r_c, d(u, u_{i-1}))$ in round i. Then u_i will compute $d(u, u_i)$ (after receiving distance estimates from all neighbors), and forwards $(r_c, d(c, u_i))$ to all neighbors including u_{i+1}. Therefore in round $i = j \leq r_c$, u_j has received the message $r_c - d(u,c)$, and can compute $d(u,c)$. Therefore this algorithm will terminate after $\max_{v \in V} r_v$ rounds. Since r_u is an exponential random variable with parameter $O(1/\ell)$, we know that maximum of these $O(n)$ exponential random variables is $O(\ell \log(n))$ with high probability. Now we need to repeat the partition algorithm $O(\log(n))$ times and will pipeline the broadcasts for different runs. Clearly, each node is in at most in $O(\log(n))$ clusters. A standard Chernoff bound in combination with Lemma 9 implies that with high probability after $O(\log(n))$ repetition of the ℓ-limited LDD($O(1/\ell)$) algorithm, for each path π with at most ℓ hops and length $w(\pi) \in [R, 2R]$, there will be a cluster C such that $\pi \subseteq C$. We then repeat this process for $O(\log(W))$ distance scales to get an ℓ-limited W-pairwise cover. ◀

As we will see, since in our hopset construction we consider different distance scales and need to compute pairwise for a fixed scale, this step takes only $O(\ell \log(n))$ rounds.

Diameter guarantee. For constructing pairwise covers, we need the diameter guarantee of $O(W \log(n))$ for all clusters. While running LDD($O(1/\ell)$) gives a diameter guarantee of $O(\ell \log(n))$ on $\hat{G}$, we note that the construction ensures that clusters have diameter $O(W \log(n))$ on G. Since we argued that every ℓ hop path with length W will fall into a cluster with high probability, the diameter guarantee of $O(\ell \log(n))$ on $\hat{G}$ will imply that the corresponding cluster in G will have length $O(W \log(n))$. More formally, for any pair of nodes there is a path with length $O(\ell \log(n))$ in $\hat{G}$. Let $\mathcal{C}$ denote the cluster that contains this path. Consider each segment of length $O(\ell)$ in $\hat{G}$ is in $\mathcal{C}$ and will be have length $O(W)$ in G (by Lemma 10), and thus there will be a path of length $O(W \log(n))$ based on weights in G in $\mathcal{C}$.

Extension to neighborhood covers. While Cohen shows that for the parallel construction of hopsets pairwise covers are enough, for the distributed implementation we need one more property: each vertex should overlap with at most $O(\log(n))$ clusters. Moreover, the algorithm used in Theorem 11 provides the stronger guarantee that there will be a cluster that contains the neighborhood of (weighted) radius W from each vertex with high probability, rather than only containing paths of length W. In other words, a similar analysis shows that with high probability an ℓ-limited neighborhood cover can be constructed in $O(\ell \log(n))$ rounds of the CONGEST model. That is, for each node u, the ℓ-limited W-neighborhood of u will be fully contained in a cluster with high probability. However, for our purposes the path covering property suffices.

5 Congested Clique Hopset Construction.

In this section we describe our main algorithm. Similar to the sequential construction described we consider different distance scales $[R, 2R)$, and handle each scale separately. In each iteration, we construct a *sparse* ℓ-limited $2R$-neighborhood cover as described in Section 4. Then the clusters will be divided into small and big clusters, and each case will be handled differently. So far the construction is similar to [8]. The key new idea is that for Congested Clique, by setting the parameters carefully we can send the topology corresponding to a small cluster to the cluster center, and build a hopset locally. Here we need to use the fact that each node is in at most $O(\log(n))$ clusters, which is a property that we get from our neighborhood cover construction. We will also need to compute pairwise distances between big clusters centers. For this step, we use the algorithm of [6] that computes $(1+\epsilon)$-multi-source shortest path from $O(\sqrt{n})$ sources (Lemma 6). We note that while during our construction we construct the denser hopsets of [6] as auxiliary structure, these extra edges will be removed at the end of each distance scale.

Finally, we use Lemma 5 to use the hopset edges added for smaller distance scales to construct the larger distance scales. For this to give us a $(1+\epsilon)$ for an arbitrary ϵ, we first let ϵ' be the error parameter. Since we use paths with error $(1+\epsilon')$ for each scale, to compute distances for the next scale, a multiplicative factor in the stretch will be added in each iteration. This means that after i iterations the error will be $(1+\epsilon')^i$. We can simply rescale the error parameter by setting $\epsilon' = O(\frac{\epsilon}{\log(n)})$ to get arbitrary error overall of $\epsilon > 0$.

Throughout our analysis w.l.o.g we assume the minimum edge weight is one. Otherwise, we can scale all the edge weights by the minimum edge weight. We also assume the aspect ratio is polynomial. Otherwise we can use reductions from previous work to reduce the aspect ratio in exchange in polylogarithmic depth (this will be preprocessing step and will not dominate the overall running time).

Algorithm 1 Congested Clique construction (ϵ, β)-hopset of size $\tilde{O}(n^{1+\frac{1}{2k}})$.

Let H_i denote the hopset edges for scale $(2^i, 2^{i+1}]$, and set $\epsilon' = O(\frac{\epsilon}{\log(n)})$.
for $(R, 2R]$, *where* $R = 2^\kappa, \log(\beta) \leq \kappa \leq O(\log(n))$, *on* $G \cup_{i \geq \log(\beta)}^{\kappa-1} H_i$ **do**
  Set $W = O(\epsilon' R/(\log n))$, and and build β-limited W-pairwise covers (by Theorem 11).
  Let $\mathcal{C}_b$ be the set of big clusters that have size at least $\sqrt{n}$ and $\mathcal{C}_s$ small clusters to have size less than $\sqrt{n}$.
  for *each* $C \in \mathcal{C}_s$ **do**
    All the nodes in C cluster send their incident edges to the center.
    The center locally computes a hopset of size $O(n^{1/2+1/2k})$ (construction of [16]) with (ϵ', β_0)-hopsets with $\beta_0 = O(\log(k)/\epsilon')^{\log(k+1)-1}$.
    The center sends the new hopset edges to the corresponding nodes (endpoints) in C.
  for *each* $C \in \mathcal{C}_b$ **do**
    Add a star by adding edges from the center node of each big cluster to every other node in that cluster (limiting exploration to $\ell = 2\beta + 1$ hops), and set the weights based on shortest path distances.
  Add an edge between any pair u_1, u_2 of centers of big clusters that are within $\ell = 2\beta + 1$ hops of each other, and set the weight to $d^\ell(u_1, u_2)$.

An overview of the algorithm is presented in Algorithm 1. By defining small clusters to have size at most $\sqrt{n}$, we have that the number of edges in each small cluster C is $O(n)$, and hence all the nodes C can send their incident edges to the cluster center in constant rounds using Lenzen's routing [18]. Then the cluster center computes a hopset with size $O(n^{1/2+1/2k})$ and hopbound $\beta_0 = O(\log(k)/\epsilon')^{\log(k)-1}$ *locally* using Huang-Pettie [16] centralized construction. The center of a small cluster C can send the edges incident to each node in that clusters. Since the size of the hopset on small clusters is always $O(n^{\frac{1}{2}+\frac{1}{2k}}) = O(n)$, this can also be done in constant time using Lenzen's routing.

As explained in Lemma 5, using hopset edges added for smaller scales we can limit all the shortest path explorations to $2\beta + 1$. So we can add the star edges, by running $2\beta + 1$ rounds of Bellman-Ford. For adding a clique between centers of large clusters, we will use the $(1 + \epsilon)$-MSSP algorithm of Censor-Hillel et al. 2019 [6] (using Lemma 6. This is possible since there are at most $O(\sqrt{n})$ nodes. We disregard all the other edges added in this step for computing these distances after the computation. We now analyze the algorithm and show that it has the properties stated in Theorem 1.

Hopbound. The hopbound of hopsets constructed for small clusters, which are based on Huang-Pettie hopsets is $\beta' = O((\log(k)/\epsilon')^{\log(k+1)-1})$. The path between a pair of nodes in each distance scale has $O(\log(n)/\epsilon')$ segments. The properties of a neighborhood cover imply that each of these segments are w.h.p. contained in one cluster. Each such segment has a corresponding path with hopbound either β' (if it is contained in a small clusters), or one single edge (if there is only one big cluster). If there is more than one big cluster center, then there is a single edge between the furthest big cluster centers on the path. Hence in the worst case all segments correspond to small clusters and will have a corresponding path of length β'. Therefore the overall hopbound is $O(\frac{\log(n)}{\epsilon'}(\frac{\log(k)}{\epsilon'})^{\log(k+1)-1})) = O(\frac{\log^2(n)}{\epsilon}(\frac{\log(n)\log(k)}{\epsilon})^{\log(k+1)-1})$.

Size. Recall that large clusters have size at least $\sqrt{n}$. The stars added for each big cluster will add $O(n\log^2(n))$ edges overall since they are consisted of unions of $O(\log(n))$ forests for each scale. The (clique) edges added between centers of big clusters will add $O(n)$ edges overall. For small clusters of size $s = O(\sqrt{n})$, we added a hopset of size $s^{1+1/k}$ (this is the guarantee we get by using Huang-Pettie hopsets), for a parameter $k \geq 2$. On the other hand, we have at most $O(\frac{n}{s})$ clusters of size within $[s, 2s]$. Therefore we can estimate the overall number of edges added for these small clusters in each scale by summing over different values $s \in [2^r, 2^{r+1})$ for small clusters as follows:

$$\sum_{s\in[\sqrt{n}]} O(\frac{n}{s}\cdot s^{1+1/k}) = \sum_{r=1}^{\log(\sqrt{n})} O(\frac{n}{2^r}\cdot(2^r)^{1+1/k}) = \sum_{r=1}^{\log(\sqrt{n})} O(n\cdot 2^{\frac{r}{k}}) = O(n^{1+\frac{1}{2k}}).$$

Therefore, the overall size for all scales is $O(n^{1+\frac{1}{2k}}\log(n) + n\log^2(n))$.

Stretch. Fix a distance scale $(R, 2R], R = 2^k$ and consider a pair of nodes $u, v \in V$ where $d(u, v) \in (R, 2R]$. If $R \leq \log\beta$, since we assumed the minimum edge weight is one, this implies that the shortest path has at most $O(\beta)$ hops and no more edges is needed for this pair. Otherwise, let π be the shortest path between u and v. We look at three different cases and show $\tilde{d}(u, v) = (1 + \epsilon')d(u, v) + O(W\log n)$.

First consider the case where all the clusters on the shortest path between u and v are small clusters. In this case, we have replaced each segment of length W with a path of stretch $(1 + \epsilon')$. By the triangle inequality, overall we get a $(1 + \epsilon')$-stretch. Next, consider a case where there is a single large cluster on this path. The segment corresponding to this single cluster will just add a single additive $W\log n = \epsilon' R$ cost to our distance estimate.

Final case is when there are more than one large clusters. Consider the two furthest large clusters (based on their centers) on π, and let their centers be x and y. We have added one single edge within $(1+\epsilon')$-stretch of $d(x,y)$ that covers the whole segment between these two centers. Therefore, we have shown that all segments of π have a corresponding path within $(1+\epsilon')$-stretch. As argued, this implies that each scale incurs a multiplicative factor of $(1+\epsilon')$ in the stretch, and thus by setting $\epsilon' = O(\epsilon/\log(n))$, and since we assumed that the weights are polynomial we can get $(1+\epsilon)$-stretch for all scales.

Round Complexity. For each of the $O(\log(n))$ distance scales, it takes $O(\beta\log^2(n))$ rounds to compute $2\beta+1$-limited neighborhood covers (Lemma 11). Once the covers are constructed for small clusters we need to run a Bellman-Ford with $O(\beta)$ hops from the center of each big cluster and since each node may overlap with at most $O(\log(n))$ clusters this phase takes $O(\beta\log(n))$ (each node can pipeline the computation over the clusters it overlaps with). For small clusters, we argued that in $O(1)$ rounds (using Lenzen's routing) the whole small cluster topology can be sent to the cluster center, and after local computation another $O(1)$ rounds will be enough for cluster center to send back the new hopset edges to the destination node. Finally, using the result of [6] we can compute $(1+\epsilon')$-approximation from big cluster centers ($O(\sqrt{n})$ sources) in $O(\frac{\log^2(n)}{\epsilon'}) = O(\frac{\log^3(n)}{\epsilon})$ time. Therefore the overall running time is $O(\beta\log^2(n))$.

Application to multi-source queries. We can now combine our hopset construction with Lemma 6 (source detection algorithm of [6]) to show that we can compute queries from $O(\sqrt{n})$ sources in $O(\beta)$ time, by maintaining a sparse hopset of size $\tilde{O}(n^{1+\frac{1}{2k}})$, while [6] has to store a hopset of size $\tilde{O}(n^{3/2})$. Corollary 2 follows from this observation.

6 Massively Parallel Hopsets and MSSP

In this section, we argue that in a variation of the MPC model where the overall memory is $O(mn^{1/2})$ we can construct hopsets with small hopbound efficiently, and this in turn gives us a fast algorithm for multi-source shortest path in this case. This result relies on an observation made in [9], stating that the PRAM hopset constructions (e.g. [8], [11]) that use $O(m\alpha)$ processors with depth t can be implemented in MPC, even when the memory per machine is strictly sublinear, in $O(t)$ rounds if we assume that the overall memory available is $O(m\alpha)$. Once a (β,ϵ)-hopset is constructed, the Bellman-Ford subroutine described in [9] can be used to compute $(1+\epsilon)$-stretch distances from $O(\sqrt{n})$ nodes to all other nodes in V.

Results of [9] are based on hopsets of [11], and their constructions may use less overall memory in general, but they get a worse hopbound than ours in the special case that the total memory is $\tilde{\Omega}(m\sqrt{n})$. In this case, we get an improved hopbound of $O(\log^2(n)/\epsilon)$, whereas their result gives a hopbound of $O((\log(n)/\epsilon)^3)$. In particular, we use the PRAM hopset construction of [8] (instead of [11]), which can be simulated in the MPC model with strictly sublinear memory per machine (using a reduction of [13]) to construct hopsets with hopbound $O(\log^2(n)/\epsilon)$. The only difference between our construction and [8] is using a faster algorithm for constructing ℓ-limited pairwise covers based on the algorithm of [20]. First we note that our ℓ-limited W-neighborhood cover construction can be constructed in MPC based on a very similar algorithm and analysis as in Section 4. This step can be done only using $O(m\log^2(n))$ overall memory (or $O(m\log(n))$ memory for a single-scale) in $O(\ell\log(n))$ rounds. Observe that the construction of W-neighborhood covers for different scales $[W,2W]$ can all be done in parallel with an extra logarithmic overhead in the *total*

memory. Similarly since each of the low diameter partitions are independent, the repetitions of the LDD algorithm can also be parallelized. This result is also implied by results in [20], combined with goodrich [13]. We have,

▶ **Lemma 12.** *There is an algorithm that runs in $O(\frac{\ell}{\gamma} \cdot \log(n))$ rounds of MPC and w.h.p. computes an ℓ-limited W-neighborhood cover, where memory per machines is $O(n^\gamma), 0 < \gamma \leq 1$ and the overall memory is $O(m \log^2(n))$.*

$(1+\epsilon)$-MSSP. Given a pairwise cover, assuming that in MPC we have $\tilde{O}(mn^{1/2})$ total memory, we can construct a $(\log^2(n)/\epsilon, \epsilon)$-hopset of size $O(n^{3/2} \log(n))$. This hopset is a special case of hopsets of [8]: we add a clique for small clusters, a star centerd at each big cluster, and a clique between big cluster centers. As stated, the main difference in our algorithm is that we use the algorithm of Lemma 12 for constructing pairwise covers, rather than the algorithm of [7]. This leads to a construction time of $O(\beta \log(n))$, whereas a direct reduction from [8] would have construction time of $O(\beta \log^3(n))$, which is how long it takes to construct their limited pairwise covers. Hence combining Lemma 12 with simulating the PRAM construction of [8], and the Bellman-Ford primitives described in [9], we can construct a hopset of size $O(n^{3/2} \log(n))$ in $O(\beta \log(n))$ time with hopbound $\beta = O(\log^2(n)/\epsilon)$.

▶ **Theorem 13.** *Given an undirected weighted graph G, and parameters $\epsilon > 0, 0 < \gamma \leq 1$, we can w.h.p. construct an (β, ϵ)-hopset of size $O(n^{3/2} \log(n))$ in $O(\frac{\log^3(n)}{\gamma\epsilon})$ rounds of MPC, using $O(n^\gamma)$ memory per machine, and the overall memory of $O(mn^{1/2})$ (i.e. there are $O(mn^{1/2-\gamma})$ machines), where hopbound is $\beta = O(\frac{\log^2(n)}{\gamma\epsilon})$.*

The analysis is very similar to the arguments in previous sections and previous work. Similarly, for $(1+\epsilon)$-MSSP we get,

▶ **Theorem 14.** *Given an undirected weighted graph G, after a preprocessing step of $O(\frac{\log^3(n)}{\gamma\epsilon})$ rounds, we can w.h.p. compute $(1+\epsilon)$-multi source shortest path queries from $O(n^{1/2})$ sources in $O(\frac{\log^2(n)}{\gamma\epsilon})$ rounds of MPC, when the memory per machine is $O(n^\gamma), 0 < \gamma \leq 1$, and the overall memory required for preprocessing is $O(mn^{1/2})$.*

At a high-level since we have overall memory of $\tilde{O}(mn^{1/2})$, to each node u, we can assign a block of memory of size $O(\deg(u).n^{1/2})$. Then using aggregations primitives (e.g. see [9]), we can store and update the distances from up to $O(n^{1/2})$ sources. Therefore given a hopset with hopbound $O(\log^2(n)/\epsilon)$, we can compute distances from $O(n^{1/2})$ sources by running parallel Bellman-Ford.

References

1. Amir Abboud, Greg Bodwin, and Seth Pettie. A hierarchy of lower bounds for sublinear additive spanners. *SIAM Journal on Computing*, 2018.
2. Baruch Awerbuch and David Peleg. Sparse partitions. In *Proceedings of the Symposium on Foundations of Computer Science*. IEEE, 1990.
3. Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. In *Proceedings of the Symposium on Principles of Database Systems*. ACM, 2013.
4. Ruben Becker, Andreas Karrenbauer, Sebastian Krinninger, and Christoph Lenzen. Near-Optimal Approximate Shortest Paths and Transshipment in Distributed and Streaming Models. In *31st International Symposium on Distributed Computing*, 2017.
5. Soheil Behnezhad, Mahsa Derakhshan, and MohammadTaghi Hajiaghayi. Brief Announcement: Semi-MapReduce Meets Congested Clique. *arXiv preprint arXiv:1802.10297*, 2018.

6 Keren Censor-Hillel, Michal Dory, Janne H Korhonen, and Dean Leitersdorf. Fast Approximate Shortest Paths in the Congested Clique. In *Proceedings of the Symposium on Principles of Distributed Computing*. ACM, 2019.

7 Edith Cohen. Fast algorithms for constructing t-spanners and paths with stretch t. *SIAM Journal on Computing*, 1998.

8 Edith Cohen. Polylog-time and near-linear work approximation scheme for undirected shortest paths. *Journal of the ACM (JACM)*, 2000.

9 Michael Dinitz and Yasamin Nazari. Massively Parallel Approximate Distance Sketches. *OPODIS*, 2019.

10 Michael Elkin and Ofer Neiman. Near-Optimal Distributed Routing with Low Memory. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*. ACM, 2018.

11 Michael Elkin and Ofer Neiman. Hopsets with constant hopbound, and applications to approximate shortest paths. *SIAM Journal on Computing*, 2019.

12 Michael Elkin and Ofer Neiman. Linear-Size Hopsets with Small Hopbound, and Constant-Hopbound Hopsets in RNC. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 2019.

13 Michael T Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, searching, and simulation in the mapreduce framework. In *Proceedings of the International Symposium on Algorithms and Computation*. Springer, 2011.

14 James W Hegeman and Sriram V Pemmaraju. Lessons from the congested clique applied to MapReduce. *Theoretical Computer Science*, 2015.

15 Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. A deterministic almost-tight distributed algorithm for approximating single-source shortest paths. In *Proceedings of the Symposium on Theory of Computing*. ACM, 2016.

16 Shang-En Huang and Seth Pettie. Thorup–Zwick emulators are universally optimal hopsets. *Information Processing Letters*, 2019.

17 Philip N Klein and Sairam Subramanian. A randomized parallel algorithm for single-source shortest paths. *Journal of Algorithms*, 1997.

18 Christoph Lenzen. Optimal deterministic routing and sorting on the congested clique. In *Proceedings of the ACM Symposium on Principles of Distributed computing*. ACM, 2013.

19 Zvi Lotker, Boaz Patt-Shamir, Elan Pavlov, and David Peleg. Minimum-weight spanning tree construction in $O(loglogn)$ communication rounds. *SIAM Journal on Computing*, 2005.

20 Gary L Miller, Richard Peng, Adrian Vladu, and Shen Chen Xu. Improved Parallel Algorithms for Spanners and Hopsets. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures*. ACM, 2015.

21 Gary L Miller, Richard Peng, and Shen Chen Xu. Parallel graph decompositions using random shifts. In *Proceedings of the ACM Symposium on Parallelism in algorithms and architectures*. ACM, 2013.

22 Danupon Nanongkai. Distributed approximation algorithms for weighted shortest paths. In *Proceedings of the ACM Symposium on Theory of Computing*. ACM, 2014.

23 Merav Parter and Eylon Yogev. Low congestion cycle covers and their applications. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, 2019.

24 Mikkel Thorup and Uri Zwick. Approximate distance oracles. *Journal of the ACM (JACM)*, 2005.

Massively Parallel Approximate Distance Sketches

Michael Dinitz
Johns Hopkins University, Baltimore, MD, United States
mdinitz@cs.jhu.edu

Yasamin Nazari
Johns Hopkins University, Baltimore, MD, United States
ynazari@jhu.edu

Abstract

Data structures that allow efficient distance estimation (distance oracles, distance sketches, etc.) have been extensively studied, and are particularly well studied in centralized models and classical distributed models such as CONGEST. We initiate their study in newer (and arguably more realistic) models of distributed computation: the Congested Clique model and the Massively Parallel Computation (MPC) model. We provide efficient constructions in both of these models, but our core results are for MPC. In MPC we give two main results: an algorithm that constructs stretch/space optimal distance sketches but takes a (small) polynomial number of rounds, and an algorithm that constructs distance sketches with worse stretch but that only takes polylogarithmic rounds.

Along the way, we show that other useful combinatorial structures can also be computed in MPC. In particular, one key component we use to construct distance sketches are an MPC construction of the hopsets of [9]. This result has additional applications such as the first polylogarithmic time algorithm for constant approximate single-source shortest paths for weighted graphs in the low memory MPC setting.

2012 ACM Subject Classification Theory of computation → Massively parallel algorithms

Keywords and phrases Distance Sketches, Massively Parallel Computation, Distance Oracles, Single-Source Shortest Paths

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.35

Related Version The full version of this paper is available at `https://arxiv.org/pdf/1810.09027.pdf`.

Funding This work supported is in part by NSF awards CCF-1464239 and CCF-1909111.

Acknowledgements The authors are thankful to Goran Zuzic for helpful discussions.

1 Introduction

A common task when performing graph analytics is to compute distances between vertices. This has motivated the study of shortest path algorithms in essentially every interesting model of computation. We focus on two models which correspond to modern big-data graph analytics: Congested Clique [18] and Massively Parallel Computation (MPC) [3]. The MPC model in particular has recently received significant attention, as it captures many modern data analytics frameworks such as MapReduce, Hadoop, and Spark. So since these are important models of distributed storage and computation, and computing distances in graphs is an important primitive, we have an obvious question: in MPC or Congested Clique, can we compute distances between nodes sufficiently quickly to support important graph analytics?

While one side effect of our techniques is indeed a state of the art algorithm for shortest paths in MPC, the focus of this paper is on getting around the limitations of these models by allowing preprocessing of the (distributed) graph. We will first spend some time building a data structure known as *approximate distance sketches* (or an approximate distance oracle), which will then let us (approximately) answer any distance query using only 0, 1, or 2 rounds

© Michael Dinitz and Yasamin Nazari;
licensed under Creative Commons License CC-BY
23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller; Article No. 35; pp. 35:1–35:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

of network communication (depending on the precise model). Thus after this preprocessing, anyone who is interested in analyzing the massive graph has access to approximate distances essentially for free, making this a powerful tool for distributed graph analytics. Moreover, rather than inventing a brand new structure, we show that we can repurpose centralized data structures (in particular the Thorup-Zwick oracle [25]) by computing them efficiently in these new distributed models. And since our algorithms are derived from centralized data structures we even allow for extremely efficient computation in addition to efficient communication.

So our focus is on how to compute these data structures efficiently, since once they are computed distance estimates become fast and easy. We show that in both the Congested Clique and the MPC models, we can compute oracles/sketches which essentially match the best centralized bounds in time that is only a small polynomial. In MPC, we can go even further and compute slightly suboptimal sketches in time that is only polylogarithmic. So while computing the data structure is still somewhat expensive, it is far more efficient than trivial approaches, and once it is computed, the analyst can receive approximate distances extremely quickly, allowing for low amortized cost or just the ability to do exploratory analysis without constantly waiting for expensive distance queries to complete.

Distance Oracles and Sketches. Even in many centralized applications, the time it takes to compute exact distances in graphs is undesriable, and similarly the memory that it would take to store all $\binom{n}{2}$ distances is also undesirable. This motivated Thorup and Zwick [25] to define the notion of an *approximate distance oracle*: a small data structure which can quickly report an approximation of the true distance for any pair of vertices. In other words, by spending some time up front to compute this data structure (known as the *preprocessing* step) and then storing it (which can be done since the structure is small), any algorithm used in the future can quickly obtain provably accurate distance estimates.

More formally, an approximate distance oracle is said to have *stretch* t if, when queried on $u, v \in V$, it returns a value $d'(u, v)$ such that $d(u, v) \leq d'(u, v) \leq t \cdot d(u, v)$ for all $u, v \in V$, where $d(u, v)$ denotes the shortest-path distance between u and v. The important parameters of an approximate distance oracle are the size of the oracle, the stretch, the query time, and the preprocessing time. For any constant k, Thorup and Zwick's construction (in the sequential setting) has expected size $O(kn^{1+1/k})$, stretch $(2k-1)$, query time $O(k)$, and preprocessing time $O(kmn^{1/k})$, where $n = |V|$ and $m = |E|$.

Since [25], there has been a large amount of followup work on improving the achievable tradeoffs, such as achieving query time of $O(1)$ with size $O(n^{1+1/k})$ [26, 7] or giving more refined bounds [20, 21]. However, with the notable exception of a very interesting construction due to Mendel and Naor [19], the vast majority of followup work has essentially been refinements and improvements to the approach pioneered by Thorup and Zwick. Thus understanding the Thorup-Zwick distance oracle is an important first step to understanding the limits and possibilities of distance oracles, and showing how to construct the Thorup-Zwick oracle in different computational models gives almost state-of-the-art bounds while also developing the basic tools and framework needed to design more sophisticated structures.

Importantly, the Thorup-Zwick distance oracle has the additional property that the data structure can be "broken up" into n pieces, each of size $O(kn^{1/k} \log n)$, so that the estimate $d'(u, v)$ can be computed just from the piece for u and the piece for v (the rest of the structure is unnecessary). These are called *distance sketches* or *distance labelings*, and motivated Das Sarma et al. [24] to initiate the study of Thorup-Zwick distance sketches in distributed networks, and in particular in the CONGEST model of distributed computing [22].

Models. As mentioned, in modern graph analytics we usually abstract away the communication graph by assuming that the datacenter storing the graph is sufficiently well-provisioned. This motivated two different but related models of distributed computation: Congested Clique [22] and MPC [3]. In the Congested Clique model an input graph of $G = (V, E)$ is given, and initially each node $v \in V$ only knows its incident edges. However, the underlying communication graph is an undirected clique, and in each round every node can send a message of $O(\log n)$ bits to any other node. This model was introduced by [22], and has been studied extensively in recent years. The second model that we consider is the *Massively Parallel Computation*, or MPC model. This model was introduced by [3] to model MapReduce and other realistic distributed settings, and is more general than earlier abstractions of MapReduce proposed by [15] and [12]. In this model there is an input of size N which is arbitrarily distributed over N/S machines, each of which has $S = N^\epsilon$ memory for some $0 < \epsilon < 1$. In the standard MPC model, every machine can communicate with every other machine in the network, but each machine in each round can have total I/O of at most S. Specifically, for graph problems the total memory N is $O(|E|)$ words. The low memory setting is the more challenging (but arguably more realistic) setting in which each machine has has $O(n^\gamma), \gamma < 1$ memory, where $n = |V|$, which we denote by MPC(n^γ). We also make the common assumption (e.g. [23, 3]) that machines have unique IDs that other machines can use for direct communication.

1.1 Our Results

In this paper we initiate the study of distance oracles and sketches in two popular computational models for "big data": Congested Clique and MPC. In addition, we show that our techniques can be used to give the first sublinear algorithm (and in fact polylogarithmic) for approximate single-source shortest paths for weighted graphs in (low memory) MPC, and moreover can be applied in straightforward ways to non-distributed models such as the streaming setting. We discuss our results for each model in turn. At a high level, Congested Clique turns out to be relatively easy: we can essentially just combine the known CONGEST algorithm [24] with a slightly modified hopset construction. For MPC, the natural approach is to simulate the Congested Clique algorithm, since it is known [5] that under certain density and memory conditions, Congested Clique algorithms can be simulated in MPC. However, this simulation requires at least $\Omega(n)$ memory per machine. Our task becomes much more challenging if we allow $o(n)$ memory per machine, which we refer to as the *low memory* setting. Designing algorithms for this setting forms the bulk of this paper.

Congested Clique. Since there is no memory restriction for Congested Clique, we assume that some node in the network is the *coordinator* at which the entire distance oracle will be stored (i.e., the machine with which users will interact with the distributed system). So at query time, the user can just query the coordinator locally (avoiding all network delay) rather than initiating an expensive distributed computation. The precise statements of our results are given in the full version and are somewhat technical, so for simplicity we state one particularly interesting corollary obtained by some specific parameter settings:

▶ **Theorem 1.** *Given a weighted graph $G = (V, E, w)$, for all $k \geq 2$ and constant $\epsilon > 0$, we can construct a distance oracle with stretch $(1+\epsilon)(2k-1)$, (local) query time $O(k)$, and space $O(kn^{1+1/k}\log n)$ w.h.p. in the Congested Clique model. If $k = O(1)$, then the number of rounds for preprocessing is*[1] *$\tilde{O}(n^{1/k})$, and if $k = \Omega(\log n)$ then the number of rounds is $\tilde{O}(\log(n))$.*

Note that after a limited amount of preprocessing, distance queries can be computed without any network access whatsoever. Moreover, the computational query time is also extremely small, so these queries are extraordinarily efficient in the context of distributed algorithms. As an interesting extension, we show that the message complexity of computing this distance oracle can be reduced by adding an additional preprocessing step of computing a graph spanner.

MPC. In Section 3 we discuss the MPC model, which is the heart of this paper. Since in the MPC model servers have small memory, it is impossible to fit an entire distance oracle at a single server as we did in the Congested Clique. So we instead focus on distance sketches. After the preprocessing algorithm, for each node $v \in V$, a distance sketch of size $O(kn^{1/k}\log n)$ will be stored and mapped to a machine with key v (this assumes that the memory at each server is at least $\Omega(kn^{1/k}\log n)$, which is reasonable in most settings). This means that after the preprocessing to construct these sketches, only two rounds of communication are needed for for approximating distance queries between a pair of nodes u and v: one for sending requests for the sketches of u and v and one for receiving them. We give the following result:

▶ **Theorem 2.** *Given a weighted graph $G = (V, E, w)$ with polynomial weights*[2] *and parameters $\rho \leq \gamma \leq 1, 1/k \leq \rho, 0 < \epsilon < 1$, we can construct Thorup-Zwick distance sketches with stretch $(2k-1)(1+\epsilon)$ and size $O(kn^{1/k}\log n)$ w.h.p. in $\tilde{O}(\frac{1}{\gamma} \cdot n^{1/k} \cdot \beta)$ rounds of MPC(n^γ), where $\beta = \min(O(\frac{\log n}{\epsilon})^{\log(k)+k}, 2^{\tilde{O}(\sqrt{\log n})})$. In particular, if $k = O(1)$ and ϵ is a constant, then w.h.p. we require $\tilde{O}(n^{1/k})$ rounds, and if $k = \Theta(\log n)$ then w.h.p. we require $2^{\tilde{O}(\sqrt{\log n})}$ rounds.*

In the above theorem the distance sketches have the same guarantees as the centralized Thorup-Zwick distance oracles. However, in MPC a polynomial round complexity, while possibly of theoretical interest, is generally considered not practical. So we give a different (but related) algorithm which achieves polylogarithmic round complexity, at the price of larger stretch.

▶ **Theorem 3.** *Consider a graph $G = (V, E)$ where $m = \Omega(kn^{1+1/k}\log n)$, for any $k \geq 2$. Then there is an algorithm in MPC(n^γ) (with $0 < \gamma < 1$) that constructs Thorup-Zwick distance sketches with stretch $O(k^2)$ and size $O(kn^{1/k}\log n)$ and with high probability completes in $O(\frac{k}{\gamma} \cdot (\frac{\log n \cdot \log k}{\epsilon})^{\log k + k - 1})$ rounds.*

As a side effect of our techniques (which we discuss more in Section 1.2), we immediately get an algorithm for computing approximate single-source shortest paths (SSSP) in the MPC model, which is the problem of finding the (approximate) distances from a source node to all

[1] The notation $\tilde{O}(f(n))$ stands for $O(f(n) \cdot \text{polylog}(f(n))$, e.g. it is suppressing polyloglog(n) terms in $2^{\tilde{O}(\log n)}$.

[2] This assumption can be relaxed using reduction techniques (e.g. from [9]) in exchange for extra polylogarithmic factors in the hopbound and construction time.

other nodes. Unlike in the Congested Clique, there do not seem to be any known nontrivial results for this problem in MPC. We first give an algorithm which computes a $(1+\epsilon)$-approximation in $n^{o(1)}$ time. Then we show that we can compute an $O(1)$-approximation in only polylogarithmic time, if we make an additional assumption about the density of the input graph. We will prove the following theorem in Section 3.2:

▶ **Theorem 4.** *Given a weighted undirected graph $G = (V, E, w)$ with polynomial weights, a source node $s \in V$, and $0 < \gamma \leq 1, 0 < \epsilon < 1$ we can compute $(1+\epsilon)$-approximate SSSP w.h.p. in $O(\frac{1}{\gamma}) \cdot 2^{\tilde{O}(\sqrt{\log n})}$ rounds of MPC with $\Theta(n^\gamma)$ memory per machine. Moreover, if $|E| \geq \Omega(n^{1+1/k}\log(n))$, we can compute $4k(1+\epsilon)$-approximate SSSP in $O(\frac{1}{\gamma} \cdot (\frac{\log n \cdot \log k}{\epsilon})^{\log k + k - 1})$ rounds of MPC(n^γ), where $1/k < \gamma \leq 1, k \geq 2$. In particular, for $k = O(1)$ the algorithm runs in $O(\frac{1}{\gamma} \cdot (\frac{\log n}{\epsilon})^{O(1)})$ rounds.*

Note that while the round complexity is polylogarithmic, it may still be somewhat slow for certain applications: an analyst who has to wait polylogarithmic rounds for every distance query would essentially be unable to perform any analysis which depended on large numbers of distance queries. On the other hand, our main results on distance sketches allows us to pay this round complexity only once, for constructing the sketch.

Streaming. Finally, we provide an algorithm for constructing distance oracles in the multi-pass streaming model. This is essentially a side-effect of our main results for Congested Clique and MPC, but we include it for completeness. Our general results can be found in the full version. For the specific settings of constant or logarithmic stretch, we have:

▶ **Corollary 5.** *Given a graph $G = (V, E, w)$, there exists a streaming algorithm that constructs a Thorup-Zwick distance oracle of stretch $(2k-1)(1+\epsilon)$ of size $O(kn^{1+1/k}\log n)$ w.h.p. and expected space $O(n^{1+1/k} \cdot \log^2 n)$, such that if $k = O(1)$, w.h.p. we require $O(\log^k n)$ passes , and if $k = \Omega(\log n)$, w.h.p. we require $2^{\tilde{O}(\sqrt{\log n})}$ passes.*

Note that in case of $k = \Omega(\log n)$ we are in the so-called *semi-streaming* setting in which the total memory used is $O(n \cdot \text{polylog } n)$.

1.2 Our Techniques

Our main approach is to combine constructions of *hopsets* with efficient distributed constructions of Thorup-Zwick distance oracles/sketches. In particular, Das Sarma et al. [24] showed that Thorup-Zwick sketches could be computed in the CONGEST model, but the time depended on the graph diameter. So all that we really need to do is to reduce the diameter of the graph, since any CONGEST algorithm also works in the Congested Clique. This is what hopsets do: we discuss them in more detail in Section 2.2, but informally they allow us to reduce the diameter of the graph while preserving distances by adding in a carefully chosen set of weighted "shortcut" edges. Hopset constructions for the Congested Clique were given by Elkin and Neiman [9] (and more recently by[6]) so for Congested Clique we can essentially just combine result of [9] (or [6]) with [24] to get our result (modulo a small number of technicalities).

Moving to MPC introduces some significant technical difficulties, particularly when the space per machine is $o(n)$. Neither [24] nor [9] are written with MPC in mind, so we cannot simply "black-box" them as we could (mostly) in the Congested Clique. However, not surprisingly, both [24] and [9] use as a fundamental primitive a "restricted" version of the classical Bellman-Ford shortest-path algorithm that ends early, and it turns out that implementing this restricted Bellman-Ford is the main (although not the only) technical hurdle in adapting both of them to the MPC model.

When implementing restricted Bellman-Ford in low-memory MPC, the main difficulty is that since the memory at each server is $o(n)$, a single server cannot "simulate" a node in Bellman-Ford. It takes many machines to store the edges incident on any particular node, so we need to show that it is possible for many machines to simulate a single node in MPC without too much overhead. We show that this is indeed possible: Bellman-Ford and related algorithms can be implemented in low-memory MPC with very little additional overhead. Once we develop this tool, we argue that the hopsets of [9] can be constructed in low-memory MPC with essentially the same complexity as in the Congested Clique. Our implementation of Bellman-Ford and this hopset construction, as well as a few other primitives we develop for low-memory MPC (e.g., finding minimum or broadcasting on a range of machines), may be of independent interest.

Even after using hopsets, we would still need polynomial time for constructing constant stretch distance sketches. We overcome this issue and improve the running time using two ideas. First, we show that by relaxing the model to allow small additional total memory (either through extra space per machine or additional machines), we can run our algorithms in polylogarithmic number of rounds. So we just need to argue that there is a way of obtaining extra memory without actually changing the model assumptions. This is our second idea: by constructing a spanner we can sparsify the graph while keeping the memory per machine and number of machines the same. Thus from the perspective of the spanner, it will appear that we do indeed have "extra" memory. The idea of sparsifying the input to obtain extra resources has already proved to be powerful in related contexts (for example, [11] recently used spanners to give a work-efficient PRAM metric embedding algorithm). To the best of our knowledge, though, this idea has not yet appeared in the MPC graph algorithms literature.

1.3 Related Work

Distributed constructions of distance oracles and sketches have been studied extensively in the CONGEST model [24, 17, 10]. All of these algorithms have running times dependent on the graph diameter, while our algorithms run in time independent of the graph diameter. To the best of our knowledge, constructing distance oracles/sketches has not previously been studied for the Congested Clique or the MPC model. Similarly, hopsets have been used extensively in various models of computation for solving approximate SSSP ([14, 9]). Our result on hopset construction in low memory MPC also gives the first (approximate) SSSP algorithm in this model for weighted graphs (in Congested Clique there are more results known [9, 14, 4, 6], but these do not translate obviously to MPC when there is sublinear memory per machine). In a recent result, [6] gave an efficient Congested Clique algorithm that constructs hopsets of size $\tilde{O}(n^{3/2})$ with hopbound $O(\log^2(n)/\epsilon)$. Their hopsets are a special case of hopsets of [9]. In the full version, we explain how their algorithm applies to our Congested Clique result.

In the PRAM model, shortest path computation is well studied (e.g. [8, 9]), and it is known that many PRAM algorithms can be simulated in the MPC model ([15, 12]). However, most of these algorithms use $\omega(|E|)$ number of processors, in which case the simulations of [15] and [12] do not directly apply as they assume that the number of processors is at most the input size. As we argue in Section 3.1 we will still utilize an extension of this simulation. Another recent result for APSP in MapReduce by [13] also has the same drawback of using $\omega(n^2)$ processors. Result of [13] is based on matrix multiplication techniques, which are also well-studied in the PRAM model for computing APSP.

Finally, we note that distance problems have also been studied in related models such as the k-machine model ([16]). In this model [16] shows a low bound of $\Omega(n/k)$ for computing MST and shortest path trees, where k is the number of machines. To the best of our knowledge, the type of distance sketches that we consider here are not studied in the k-machine model.

2 Preliminaries and Notation

2.1 Notation

In a given weighted graph $G = (V, E)$, we denote the (weighted) distance between a pair of nodes $u, v \in V$ by $d_G(u, v)$. We may drop the subscript G when there is no ambiguity. We define the h *hop-restricted* distance between u and v to be the weight of the shortest path between u and v that uses at most h hops and denote this by $d^h(u, v)$.

We will denote the set of neighbors of a node $v \in V$ by $N(v)$. In a weighted graph G, we define the *shortest-path diameter* of G, denoted by Λ, to be the maximum over all $u, v \in V$ of the number of edges in the shortest $u - v$ path (so if the graph is unweighted this is the same as the diameter, but in weighted settings it can be larger than the unweighted diameter). Finally, a t-spanner of G is simply a subgraph which preserves distances up to a multiplicative t factor.

2.2 Algorithmic Building Blocks

In this section we describe the algorithms of [25], [24] and [9], that we will use in next section.

Thorup-Zwick Distance Oracle. In this section, we briefly describe the centralized construction of the well-known Thorup-Zwick distance oracle [25]. Given an undirected weighted graph $G = (V, E, w)$ and $k > 1$, in the preprocessing phase of their algorithm they first create a hierarchy of subsets $A_0, A_1, ..., A_{k_1}$ by sampling from nodes of V in the following manner: set $A_0 = V$, and for $1 \leq i \leq k - 1$, add every node $v \in A_{i-1}$ to the set A_i independently with probability $n^{-1/k}$. Set $A_k = \emptyset$ and for all $u \in V$ define $d(u, A_k) = \infty$. Let $B_i(u) = \{w \in A_i : d(u, w) < d(u, A_{i+1})\}$ for all $u \in V$ and $0 \leq i \leq k - 1$, where $d(u, A_i)$ is the minimum distance between u and a node in the set A_i, and set $B(u) = \cup_{i=0}^{k-1} B_i(u)$. We also denote the node that has the minimum distance to u among all nodes in A_i by $p_i(u)$ and call this the i-center of u, and so $d(u, A_i) = d(u, p_i(u))$. The distance sketch for u consists of $\{p_i(u)\}_{i=0}^{k}$, the set $B(u)$, and the corresponding distances between these nodes and u. The distance oracle is just the union of the sketches for all $u \in V$. Thorup and Zwick showed that this data structure has size $O(kn^{1+1/k} \log n)$ w.h.p., and access to these sketches is enough for approximating distances between every pair of vertices in $O(k)$ time with stretch $2k - 1$. In all the settings we consider, after preprocessing the distance oracle/sketches, we can *locally* perform the query algorithm of [25] in $O(k)$ time.

Next, we explain a distributed construction of Thorup-Zwick distance *sketches* as described by Das Sarma et al. [24] for the CONGEST model. The sampling phase can easily be done in distributed settings. Then for finding $p_i(v), 1 \leq i \leq k$ for all nodes $v \in V$, we will do the following: in iteration i, define a virtual source node s_i, and for all nodes in $u \in A_i$ add an edge between u and s_i where $w(u, s_i) = 0$. Then we will only need to run the Bellman-Ford algorithm from s_i, and after $O(k\Lambda)$ time every node $u \in V$ knows $p_i(u)$ and $d(u, A_i)$. Finally, for all $1 \leq i \leq k$ we need to compute the distance from $w \in A_i \setminus A_{i+1}$ to all the nodes v for which $w \in B(v)$. Simply running a distributed Bellman-Ford independently from all the

sources $w \in A_i \setminus A_{i+1}$ would be slow since due to congestion limit on each edge we cannot run all these in parallel at the same time. However, [24] argue that this can be done in $O(\Lambda \cdot kn^{1/k} \log n)$ rounds in total (w.h.p), since each node v needs to forward messages in the runs of Bellman-Ford algorithm for a source w only if $w \in B(v)$. This means that, roughly speaking, each node v participates in $|B(v)| = O(kn^{1/k} \log n)$ runs of Bellman-Ford. Then by a simple round-robin scheduling scheme they show that running these Bellman-Fords for all sources in $A_i \setminus A_{i+1}$ can be done in $O(\Lambda \cdot kn^{1/k} \log n)$ without violating the congestion bound on each edge. For completeness we include a more detailed version of this algorithm in the full version.

Hopsets. For parameter $\epsilon, \beta > 0$, a graph $G_H = (V, H, w_H)$ is called a (β, ϵ)-hopset for the graph G, if in graph $G' = (V, E \cup H, w')$ obtained by adding edges of G_H, we have $d_G(u, v) \leq d^{\beta}_{G'}(u, v) \leq (1 + \epsilon) d_G(u, v)$ for every pair $u, v \in V$ of vertices. The parameter β is called the *hopbound* of the hopset.

We first give a high level overview of the (sequential) hopset construction of [9] here. In their algorithm, they consider each distance scale $(2^k, 2^{k+1}], k = 0, 1, 2, ...$ separately. For a fixed distance scale $(2^k, 2^{k+1}]$ the algorithm consists of a set of *superclustering*, and *interconnection* phases. Initially, the set of clusters is $\mathcal{P} = \{\{v\}_{v \in V}\}$. Each cluster in $C \in \mathcal{P}$ has a cluster center which we denote by r_C. The algorithm uses a sequence $\delta_1, \delta_2, ...$ of distance thresholds and a sequence $\deg_1, \deg_2, ...$ of degree thresholds that determines the sampling probability of clusters. At the i-th iteration, every cluster $C \in \mathcal{P}$ is sampled with probability $1/\deg_i$. Let S_i denote the set of sampled clusters. Now a single shortest-path exploration of depth δ_i (weighted) from the set of centers of sampled clusters $R = \{r_C \mid C \in S_i\}$ is performed. Let $C' \in \mathcal{P} \setminus S_i$ be a cluster whose center $r_{C'}$ was reached by the exploration and let r_C be the center in R closest to r'_C. An edge $(r_C, r_{C'})$ with weight $d_G(r_C, r_{C'})$ is then added to the hopset. A supercluster $\hat{C}$ with center $r_{\hat{C}} = r_C$ is now created that contains all the vertices of C and the clusters C' for which a hopset edge was added. In the next stage of iteration i, all clusters within distance $\delta_i/2$ of each other that have not been superclustered at iteration i will be interconnected. In other words, a *separate* exploration of depth $\frac{\delta_i}{2}$ is performed from each such cluster center r_C and if center of cluster C' is reached, an edge (r_C, r'_C) with weight $d_G(r_C, r_{C'})$ will be also added to the hopset. The final phase of their algorithm only consists of the interconnection phase. We denote the hopset edges added for distance scale $(2^k, 2^{k+1}]$ by H_k. For completeness, we review this algorithm in more detail and explain the exact parameters in the full version.

One important property of this hopset construction (proved in Lemma 3.3 of [9]) that we will need for our analysis in Section 3) is the following:

▶ **Lemma 6** ([9]). *In the i-th iteration of a given distance scale $(2^k, 2^{k+1}]$, for each node $v \in V$, w.h.p. the number of explorations of interconnection phase that visit v is at most $O(deg_i \cdot \log n)$, where deg_i is the sampling probability of the superclustering phase.*

Now we turn our attention to efficient construction of hopsets in distributed settings also proposed by [9]. Note that each superclustering phase can be performed by a distributed Bellman-Ford exploration of depth δ_i. For an interconnection phase, a separate distributed Bellman-Ford explorations of depth $\delta_i/2$ from cluster centers is performed. These Bellman-Ford algorithms can easily be implemented sequentially, however, in distributed settings, $O(n)$ rounds may be needed for each of the explorations of the larger scales. To overcome this issue, [9] propose to use the hopsets $\cup_{\log \beta - 1 < j \leq k-1} H_j$, for constructing hopset edges H_k. More precisely, they observe that for any pair of nodes with distance less than 2^{k+1},

hopsets $\cup_{\log\beta-1<j\leq k-1}H_j$ provide a $(1+\epsilon)$-stretch approximate shortest path with $2\beta+1$ hops between these pair of nodes. In other words, it is enough to run each Bellman-Ford exploration only for $O(\beta)$ rounds.

3 Distance Sketches in Massively Parallel Computation Model

In this section we will focus on the MPC model. First we provide MPC algorithms for constructing distance sketches that have the same guarantees (with respect to the stretch/size tradeoff) as the centralized construction of Thorup-Zwick that run in polynomial (or slightly subpolynomial) time. Then in Section 3.1 we show how we can bring down the running time to polylogarithmic in exchange for a loss in accuracy.

First, we note that it is known from [5] that for *dense graphs* with $O(n^2)$ edges every Congested Clique algorithm (in which nodes use local memory of $O(n)$) can be implemented in the MPC(n) model. Therefore, when memory per machine is $\Omega(n)$ and the graph is dense all our Congested Clique results also hold, except that we store the distance sketches rather than a central distance oracle. The more interesting case is when memory per machine is strictly sublinear in n. For the rest of this section we will turn our attention to the case where the memory is n^γ, where $0<\gamma\geq 1$ (i.e., strictly sublinear). For simplicity we assume that we can store the sketches in a single machine. Namely, we require $\tilde{O}(n^{1/k})$ memory per machine for stretch $O(k)$ distance sketches. This assumption can be relaxed (and in exchange the query algorithm will take $O(k)$ rounds instead of 2 rounds).

One main subroutine that we need is the *restricted Bellman-Ford* algorithm. We then need to run many instances of this algorithm in parallel and handle other technicalities both for constructing hopsets, and then the distance sketches. First, we require following subroutines that will allow us to simulate one round of Bellman-Ford in MPC(n^γ):

Sorting [12]. Given a set of N comparable items, the goal is to have the items sorted on the output machines, i.e. the output machine with smaller ID holds smaller items.

Indexing [1]. Suppose we have sets $S_1, S_2, ..., S_k$ of N items stored in the system. The goal is to compute a mapping f such that $\forall i\in[k], x\in S_i$, x is the $f(S_i,x)$-th element of S_i. After running this algorithm the tuple $(x, f(S_i,x))$ is stored in the machine that stores x.

Find Minimum (x,y). Finds the minimum of N values stored over a contiguous set of machines given ID x of the first machine and ID y of the last machine.

Broadcast (b,x,y). Broadcasts a message b to a contiguous group of machines given ID x of the first machine and ID y of the last machine.

The sorting and indexing subroutines can be performed in $O(1/\gamma)$ rounds of MPC(n^γ) ([1, 12]). We argue that we can solve the Find Minimum and Broadcast problems also in $O(1/\gamma)$ rounds of MPC(N^γ) in the following theorem. At a high-level we use an *implicit* aggregation tree of depth $O(\log_{N^\gamma} N)=\frac{1}{\gamma}$.

▶ **Theorem 7.** *Given N items over a contiguous range of machines x to y, subroutines Find Minimum(x,y) can be implemented in $O(1/\gamma)$ rounds of MPC(N^γ). Moreover, the subroutine Broadcast(x,y) can also be implemented in $O(1/\gamma)$ rounds of MPC(N^γ).*

Proof. We will first define a rooted *aggregation tree* $\mathcal{T}$ with branching factor N^γ where the machines $M_x, ..., M_y$ are placed at the leaves (here M_x denotes the machine with ID x). W.l.o.g assume that the machines in this range have increasing and sequential IDs. Note that we don't need to store this tree explicitly, and we only need each node to know its parent. Consider level ℓ of the tree (leaves have $\ell=0$). Each node in this level is a machine associated with the label ℓ. For each node in level $\ell-1$ that has the i-th machine in its

subtree, we set as its parent $M_{p(i,\ell)}$ where $p(i,\ell) = x + \lfloor \frac{i}{N^{\ell\gamma}} \rfloor$. Thus each machine can compute its parent given the label ℓ. Similarly, each machine can compute the indices of its children (as a range). In other words, at each level ℓ, we assign each group of N^γ nodes of this tree to a parent node at level $\ell + 1$.

The algorithm Find Minimum proceeds as follows: at each round ℓ, each machine first computes minimum over its the values it knows, and then sends the outcome to the parent machine. Finally, the minimum will be computed and stored at the root machine, which may forward the value to another destination. The algorithm Broadcast will similarly use an aggregation tree, but this time it routes the message top-down. First message b is sent to the first machine M_x, and then starting from M_x in each round any machine that receives message b sends this value to all of its children, which can be determined from the machine's ID and y. Eventually all the machines at the leaves will receive b. The number of rounds each of these subroutines take are the height of the aggregation tree which is $O(\log_{N^\gamma} N) = \frac{1}{\gamma}$. ◀

Running the (restricted) Bellman-Ford algorithm in MPC is not as straightforward as it is in the Congested Clique. One challenge is that for high-degree nodes, the edges corresponding to a single node are distributed over a set of machines. Therefore, for each round of Bellman-Ford these machines must communicate for computing and updating the distance estimates. Another hurdle is the fact that since nodes have different degrees, we do not have the range in which edges corresponding to a given node are stored a priori. To overcome these challenges we need to use the described subroutines, and for that we need to perform some preprocessing to append each edge with a tuple that we will describe shortly.

We will show how we can create and maintain the following setting: Given a graph $G = (V, E)$, the goal is to store all the edges incident to each node v in a contiguous group of machines, which we denote by $M(v)$. More precisely, let $M_1, ..., M_P$, where $P = O(\frac{m}{n^\gamma})$, be the list of machines ordered by their ID, and let $v_1, ..., v_n$ be the list of vertices sorted by their ID. $M(v_i)$ consists of the i-th smallest contiguous group of machines, such that $|M(v_i)| = \lceil \frac{\deg(v_i)}{n^\gamma} \rceil$.

Throughout the algorithm, let $M_{(u,v)}$ denote the machine that stores the edge (u, v). Also, for all $u \in V$, let r_u be the first machine in $M(u)$, and for any edge $(u, v) \in E$ let $i_u(v)$ be the index of (u, v) (based on the lexicographic order) among all the edges incident to v. We need to compute and store the following information at $M_{(u,v)}$: $\deg(u), \deg(v)$, r_u, r_v, i_u, i_v (here by storing r_u we mean ID of r_u, and for simplicity we refer to $i_u(v)$ as i_u). We first explain how these labels can be computed for all edges in $O(\frac{1}{\gamma})$ rounds in the following lemma.

▶ **Lemma 8.** *Let $M_{(u,v)}$ be the machine that stores a given edge (u, v). We can create tuples of the form $((u, v), \deg(u), \deg(v), r_u, r_v, i_u, i_v)$, stored at $M_{(u,v)}$ for all edges in $O(\frac{1}{\gamma})$ rounds in MPC(n^γ), where $\gamma < 1$.*

Proof. Let $N(v)$ be the set of edges incident on node v. Without loss of generality, let us assume that both tuples of form (u, v) and (v, u) are present in the system for each edge and we assume $(u, v) \in N(u)$ and $(v, u) \in N(v)$ (note that the graph is still undirected). First, we use the indexing subroutine of [1] on the sets $\{N(v)\}_{v \in V}$ to store index i_u at $M_{(u,v)}$ and index i_v at $M_{(v,u)}$. After this step tuples of form $((u, v), w(u, v), i_u)$ are stored at $M_{(u,v)}$.

Then we sort the tuples based on edge IDs lexicographically, using sorting algorithm proposed in [12]. This will result in the setting described above in which edges incident to each node u are stored in a contiguous group of machines $M(u)$. Now in order to compute $\deg(u)$, machines will check whether they are the last machine in $M(u)$ either by scanning their local memory or communicating with the next machine. Then the last machine in

$M(u)$ sets $\deg(u)$ to the maximum index i_u it holds. This machine can also compute r_u, ID of the first machine in $M(u)$ (using $\deg(u)$), and then broadcasts $\deg(u)$ and r_u to all machines in $M(u)$. At the end of these computations, each tuple $((u,v), w(u,v), i_u)$ will be replaced by the tuple $((u,v), w(u,v), r_u, i_u, \deg(u))$. Next, we sort these tuples again but this time based on the ID of the smallest endpoint. In other words, for each edge $(u,v) \in E$, both tuples $((u,v), w(u,v), i_u, \deg(u))$ and $((v,u), w(v,u), i_v, \deg(v))$ will be at the same machine. Now we can easily merge these two tuples to create tuples of form $((u,v), w(u,v), i_u, i_v, \deg(u), \deg(v))$. ◀

After computing the tuples, we use the sorting subroutine again to redistribute the edges into the initial setting of having contiguous group of machines $M(u)$ for all $u \in V$. After these preprocessing steps, we are ready to perform updates required for the restricted Bellman-Ford algorithm. A summary of this algorithm is presented in Algorithm 1.

Algorithm 1 Restricted Bellman-Ford in MPC(n^γ).

Input : Graph $G = (V, E)$ distributed among machines $M_1, ..., M_P$ and source s.
Output : h-hop restricted distances from the source s to all nodes $u \in V$, $d^h(s,v)$.

1 Create the tuple $((u,v), i_u, i_v, r_u, r_v, \deg(u), \deg(v))$ at $M_{(u,v)}$ for each edge $(u,v) \in E$ (by Lemma 8).
2 Sort the edges lexicographically so that edges incident to v are stored in a contiguous group of machines $M(v)$ (by [12]).
3 **for** $i = 0$ *to* h **do**
4 **for** $v \in V$ **do**
5   Compute $\hat{d}(s,v)$ by finding (using Theorem 7 $\min_{u \in N(v)} \hat{d}(s,u) + w(u,v)$).
6   Broadcast updated distances to everyone in $M(v)$ (also by Theorem 7).
7   Each machine in $M_{(v,u)}$ sends $\hat{d}(s,v)$ to $M_{(u,v)}$ (located at $r_u + \lfloor \frac{i_u}{n^\gamma} \rfloor$).

▶ **Theorem 9.** *Given a graph $G = (V, E)$ and a source node $s \in V$ the restricted Bellman-Ford algorithm (Algorithm 1) computes distances $d^h(s,v)$ for all $v \in V$ in $O(\frac{h}{\gamma})$ rounds of MPC(n^γ).*

Proof. After storing the tuples $(i_u, i_v, r_u, r_v, \deg(u), \deg(v))$ at $M_{(u,v)}$ for each $(u,v) \in E$, the restricted Bellman-Ford algorithm proceeds as follows: in each round, for each node v, we first find the minimum distance estimate for v and send it to r_v. Then r_v will broadcast the minimum distance found to all the machines in $M(v)$. By Theorem 7 both of these operations take $O(1/\gamma)$ rounds. Then for each $(v,u) \in N(v)$, $M_{(v,u)}$ sends the updated distance directly to $M_{(u,v)}$, which islocated at index $r_u + \lfloor \frac{i_u}{n^\gamma} \rfloor$. All the operations for each of the h iterations of Bellman-Ford take $O(1/\gamma)$ rounds. ◀

We now need to argue that hopsets of [9] can be constructed in MPC(n^γ). We show this in the following theorem. Here we assume that the weights are polynomial in n, which is not unrealistic since in MPC the total memory is assumed to be $\tilde{O}(m)$ bits.

▶ **Theorem 10.** *For any graph $G = (V, E, w)$ with n vertices, and parameters $\rho \le \gamma \le 1, 1 \le \kappa \le (\log n)/4, 1/2 > \rho \ge 1/\kappa$ and $0 < \epsilon < 1$, there is an algorithm in MPC(n^γ) model that computes a (β, ϵ)-hopset with expected size $O(n^{1+\frac{1}{\kappa}} \log n)$ in $O(\frac{n^\rho}{\rho} \cdot \log^2 n \cdot \beta)$ rounds whp, where $\beta = O((\frac{\log n}{\epsilon} \cdot (\log \kappa + 1/\rho))^{\log \kappa + \frac{1}{\rho}})$.*

Proof. The distributed implementation of this algorithm just performs multiple restricted Bellman-Ford algorithms in each phase. Recall also that it is enough to run each of the Bellman-Ford instances only for $O(\beta)$ rounds, by using the fact that for constructing hopset edges H_k for a distance scale of $(2^k, 2^{k+1}]$, the hopsets $\cup_{\log\beta-1<j\leq k-1} H_j$ can be used recursively.

Each round of a single Bellman-Ford algorithm can be simulated in $O(\frac{1}{\gamma})$ rounds of MPC(n^γ) by running the algorithm of Theorem 9 on each node, whose edges may be distributed over multiple machines. Hence each superclustering phase can be performed in $O(\frac{\beta}{\gamma})$ rounds. But at each interconnection phase multiple separate Bellman-Fords will run from each cluster center remaining. Thus we need to argue that these runs of Bellman-Ford will not violate the memory (and IO memory) limit of each machine. This can be shown using Lemma 6, which states that for each vetex $v \in V$, w.h.p. the number of explorations of interconnection phase that visit v is at most $O(\deg_i \cdot \log n)$. In other words, each node only forwards messages to at most $O(\deg_i \cdot \log n)$ in each depth $\delta_i/2$ Bellman-Ford explorations performed for an interconnection phase. Moreover, the parameters of their construction is set so that $\deg_i = O(n^\rho)$ throughout the algorithm. Hence, each node $v \in V$ need to store and forward distance estimates corresponding to at most $O(n^\rho \log n)$ sources for $O(\log(\kappa\rho) + \frac{1}{\rho})$ iterations, and each Bellman-Ford runs for $O(\beta)$ rounds. These separate Bellman-Ford runs can be pipelined. Overall, all of the Bellman-Ford explorations can be implemented in $O(\frac{\beta}{\gamma} \cdot n^\rho \log n)$. ◀

We can now construct a hopset first and then run the distributed variant of the algorithm in Section 2.2 due to [24] for constructing the distance sketches on the new graph. The sketch of a given node v can be stored at a machine in $M(v)$.

Proof of Theorem 2. After constructing a (β, ϵ)-hopset (by setting $\kappa = k$), we store the edges added to each node v by redistributing them among machines $M(v)$ that simulate v. Let $G' = (V, E \cup H, w')$ be the graph obtained by adding hopset edges. For constructing distance sketches with stretch $2k-1$, we run the algorithm of [24] on G'. We run the restricted Bellman-Ford algorithm (Algorithm 1) in $O(\frac{\beta}{\gamma})$ rounds. Overall, $O(\frac{\beta n^\rho \log^2 n}{\rho\gamma})$ rounds are needed for the hopset construction (by Theorem 10), and $O(kn^{1/k} \log n \cdot \frac{\beta}{\gamma})$ rounds for building the distance sketches on G'. In case $k = O(1)$ we set $\rho = 1/\kappa$, and $\kappa = k$ to get $\beta = O(1)$ and total running time $\tilde{O}(n^{1/k})$. In case $k = \Theta(\log n)$, we will set $1/\kappa = \rho = \sqrt{\frac{\log\log n}{\log n}}$. ◀

3.1 Polylogarithmic Round Complexity

In this section we describe how we can modify our algorithm to run in a polylogarithmic number of rounds in exchange for increasing the stretch. We do this by first constructing a spanner, which sparsifies the graph ("shrinking" the input) and thus allows us to act as if we have "extra" total space. It turns out that this extra space is incredibly powerful, and will let us build distance sketches in polylogarithmic time. But in the end we have to pay for both the stretch of the spanner and the stretch of the sketch, so we only achieve stretch $O(k^2)$ rather than stretch $2k-1$ for sketches of size $\tilde{O}(n^{1/k})$.

There are intuitively two reasons why this extra space is so helpful. First, in MPC having extra space (or extra machines) is equivalent to having larger total communication bandwidth. This intuitively allows us to speed up the main construction algorithm by running the Bellman-Ford algorithms "in parallel". There are some technical details but it is not surprising that extra bandwidth is helpful.

The second reason why extra space is helpful is less obvious. Goodrich et al. [12] gave a powerful simulation argument, showing that PRAM algorithms can be efficiently simulated in MPC as long as the total number of processors used and the total space used by the PRAM algorithm are bounded by the size of the input. This is a very useful theorem, but the requirement that the number of processors is only the size of the input is very restrictive. For example, the state of the art PRAM algorithms for constructing hopsets use $\Omega(mn^\rho)$ processors rather than $O(m)$ (for some value ρ determined by the parameters of the hopset). It turns out to be easy to extend [12] to show that if we have extra total space, we can use that extra space and communication to simulate PRAM algorithms that use slightly more processors or space. Thus by using a spanner first to sparsify the input, we give ourselves extra space and thus the ability to efficiently simulate a wider class of PRAM algorithms (hopsets in particular).

MPC with Extra Space. First we define a variant of MPC with extra machines (and thus extra space) denoted by $\text{MPC}(S, S')$ where S is memory per machine, the number of machines is $\Theta(\frac{mS'}{S})$ and m is the total input size. This also implies the total memory available is $\Theta(mS')$ rather than $\Theta(m)$. We are first going to analyze our algorithm in this variant of MPC, and then switch back to the standard setting.

In [12] it was shown that with a small overhead PRAM algorithms can be simulated in MPC under certain assumptions on the number of processors and the memory used. We use a simple extension of their result for our new MPC variant.

▶ **Theorem 11.** *Given a PRAM algorithm using $\mathcal{P} = O(m\alpha)$ processors that runs in time $\mathcal{T}$, and uses $O(m\alpha)$ total memory at any time, this algorithm can be simulated in $O(\mathcal{T}/\gamma)$ rounds of MPC(m^γ, α), for any $0 < \gamma < 1$.*

This stronger variant of MPC also lets us extend Theorem 7 for larger message sizes. We define a generalized variant of Find Minimum that takes a collection of vectors and computes their coordinate-wise minimum, and a generalizes version of Broadcast which broadcasts a vector of messages (rather than just a single message). We get the following lemma.

▶ **Lemma 12.** *We can compute generalized Find Minimum(x, y) over N vectors of length α stored on a contiguous range of machines x to y in $O(1/\gamma)$ rounds of MPC(N^γ, α). Moreover, the generalized Broadcast$(\mathbf{b}, x, y)$ subroutine can also be implemented in $O(1/\gamma)$ rounds.*

Proof. In the new settings we have $\Theta(N^{1-\gamma} \cdot \alpha)$ machines that can be used for computation over N items in range (x, y), rather than $\Theta(N^{1-\gamma})$ machines used in Theorem 7. Therefore we can assign each coordinate to a group of $N^{1-\gamma}$ machines and then use a similar aggregation tree argument as in Theorem 7 on all the coordinates in parallel in $O(1/\gamma)$ rounds for both problems. ◀

Next, we describe how the algorithm of Theorem 2 can be modified to utilize the extra resources in $\text{MPC}(n, n^{1/k} \log n)$ to improve the round complexity. We use an argument similar to [24] with a few changes. The complete argument can be found in the full version.

▶ **Theorem 13.** *Given a graph $G = (V, E)$ with shortest path diameter Λ, there is an algorithm in MPC($n^\gamma, n^{1/k} \log n$) that runs in time $O(k\Lambda)$ w.h.p. and constructs Thorup-Zwick distance sketches of size $O(kn^{1/k} \log n)$ with stretch $2k - 1$.*

A straightforward extension of Theorem 13 implies that given a (β, ϵ)-hopset for a graph, we can compute distance sketches with stretch $(1+\epsilon)(2k-1)$ in $O(\frac{\beta}{\gamma})$ rounds of MPC$(n^\gamma, n^{1/k}\log n)$. Next, we show that in addition to proving Theorem 13, the extra memory also lets us improve the number of rounds for the hopset construction. To show this, we use a result in [9] that constructs hopsets in PRAM, which is as follows:

▶ **Theorem 14** ([9]). *For any graph $G = (V, E, w)$ with n vertices, and parameters $2 \leq \kappa \leq (\log n)/4, 1/2 > \rho \geq 1/\kappa$ and $0 < \epsilon < 1$, there is a PRAM algorithm that computes a (β, ϵ)-hopset with expected size $O(n^{1+\frac{1}{\kappa}}\log n)$ in $O(\frac{1}{\rho} \cdot \log^2 n \cdot \log\kappa \cdot \beta)$ PRAM time whp, where $\beta = O(\frac{\log n(\log\kappa + 1/\rho)}{\epsilon})^{\log\kappa + \frac{1}{\rho}}$ using $\tilde{O}((m + n^{1+1/\kappa})n^\rho)$ processors.*

We now argue that by having more space/machines, we are can implement the algorithm in Theorem 14 with the same guarantees in low-memory MPC settings. We will not discuss the details of the PRAM construction but the intuition here is similar to Theorem 13. At a high level, having more communication/memory will allows us to perform all the $\tilde{O}(n^\rho)$ Bellman-Ford explorations required in the algorithm of Theorem 14 in parallel. By setting $\rho = 1/\kappa$ in Theorem 14 and then applying the simulation in Theorem 11 we get,

▶ **Corollary 15.** *For any graph $G = (V, E, w)$, and parameters $0 < \epsilon < 1, 1/\kappa < \gamma \leq 1, \kappa \geq 2$, there is an algorithm that computes a (β, ϵ)-hopset with size $O(n^{1+\frac{1}{\kappa}}\log n)$ w.h.p. in $O((\kappa/\gamma) \cdot \log^2 n \cdot \log\kappa \cdot \beta)$ rounds of MPC$(n^\gamma, n^{1/\kappa})$, where $\beta = O(\frac{\log n(\log\kappa)}{\epsilon})^{\log\kappa+\kappa+1}$.*

Obtaining Extra Space. Our modified algorithm for MPC(n^γ) now proceeds as follows: we first construct a spanner, then construct a hopset on this spanner, and then use Theorem 13. Intuitively, by sparsifying the graph we can "buy" more memory and hence more communication. In other words, by building a spanner we can extend the results of the extra memory setting to the standard MPC setting.

There are several efficient PRAM algorithms for constructing spanners that we can simulate in MPC. We use an algorithm proposed by [2] that constructs a $(2k-1)$-spanner of size $O(kn^{1+1/k}\log n)$ with high probability. We then use Theorem 11 with $\alpha = 1$ (i.e. the original simulation of [12]) to construct the spanner in $O(\frac{k}{\gamma}\log n \log^* n)$ rounds of MPC(n^γ), and then redistribute the spanner edges (e.g., by sorting), to make the input distribution uniform over all the machines. We can now put everything together to get the polylogarithmic construction.

Proof of Theorem 3. We first construct a $4k-1$-spanner with size $O(kn^{1+\frac{1}{2k}})$. We denote this spanner by G'. Since G' has size $m' = O(n^{1+\frac{1}{2k}})$, while our total memory (and consequently overall communication bound) is still based on the original graph. Equivalently, the number of machines is $\frac{m}{n^\gamma} = \Omega(\frac{m' n^{1/2k}\log n}{n^\gamma})$ (since $m = \Omega(kn^{1+1/k}\log n)$), and therefore we are exactly in the MPC$(n^\gamma, n^{\frac{1}{2k}})$ setting, but where the input graph is G'. Then we use Corollary 15 to construct a (β, ϵ)-hopset for G' with $\beta = O(\frac{k}{\gamma} \cdot (\frac{\log n \cdot \log k}{\epsilon})^{\log k + 1 + k})$ rounds of MPC(n^γ). Finally, after adding the hopset edges to G' we use Theorem 13. The new stretch is clearly $O(k^2(1+\epsilon))$. ◀

3.2 Single-source shortest path

In various models (such as PRAM, CONGEST and Congested Clique) hopsets are used for solving shortest path problems (e.g. [8, 14, 9]), and thus it is natural to see how they can be used for this application in the MPC model. In particular, we discuss application of Theorem 10 in solving the (approximate) single-source shortest path problem. As stated earlier, while

this problem is well-studied in many distributed models, including the Congested Clique model, we are not aware of any non-trivial results for this problem in the low memory MPC setting.

▶ **Theorem 16.** *Given a weighted undirected graph $G = (V, E, w)$, a source node $s \in V$, and $0 < \gamma \leq 1, 0 < \epsilon < 1$ we can compute $(1+\epsilon)$-approximate distances from s to all nodes in V w.h.p. in $O(\frac{1}{\gamma}) \cdot 2^{\tilde{O}(\sqrt{\log n})}$ rounds of MPC with $\Theta(n^\gamma)$ memory per machine.*

Proof. We first construct a hopset using Theorem 10 by setting $\rho = \sqrt{\frac{\log n}{\log \log n}}$, and $\kappa = \Theta(\log n)$. This will let us build a hopset with hopbound $2^{\tilde{O}(\log n)}$ in time $O(\frac{1}{\gamma}) \cdot 2^{\tilde{O}(\log n)}$. We then run the restricted Bellman-Ford algorithm (Algorithm 1) in $O(\frac{1}{\gamma}) \cdot 2^{\tilde{O}(\sqrt{\log n})}$ rounds of MPC(n^γ). The idea behind this choice of parameters is the following: any attempt to improve the running time by getting a smaller hopbound (e.g. constant) will increase the time required to construct the hopset. In other words, this choice of parameters will make the time required for preprocessing (construction of the hopset) almost the same as the time required for running the Bellman-Ford algorithm. ◀

Finally, we show that we can used the technique in Section 3.1 to find constant approximation to single source shortest path in polylogarithmic time for graphs with a certain density. In particular, by first constructing a spanner and then using Corollary 15, we can also solve $4k(1+\epsilon)$-approximate SSSP (for any $2 \leq k \leq O(\log n)$) on any graph with $m = \Omega(n^{1+1/k} \log n)$ edges in fewer number of rounds. After constructing a $4k-1$-spanner, we construct a (β, ϵ)-hopset for an appropriate hopbound β using the extra space and then run a single restricted Bellman-Ford (Algorithm 1) from the source in $O(\beta/\gamma)$ rounds of MPC(n^γ). By setting $\kappa = k$ we get,

▶ **Corollary 17.** *For any graph $G = (V, E, w)$ with n vertices, $m = \Omega(n^{1+1/k})$ edges, and $0 < \epsilon < 1, 1/k < \gamma \leq 1, k > 2$, and a source node $s \in V$, there is an algorithm that w.h.p. finds a $4k(1+\epsilon)$-approximation of shortest path distance from s to all nodes in $O(\frac{1}{\gamma} \cdot (\frac{\log n \cdot \log k}{\epsilon})^{\log k + k + 1})$ rounds of MPC(n^γ). In particular, for $k = O(1)$ the algorithm runs in $O(\frac{1}{\gamma} \cdot (\frac{\log n}{\epsilon})^{O(1)})$ rounds.*

References

1. Alexandr Andoni, Zhao Song, Clifford Stein, Zhengyu Wang, and Peilin Zhong. Parallel graph connectivity in log diameter rounds. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*. IEEE, 2018.
2. Surender Baswana and Sandeep Sen. A simple and linear time randomized algorithm for computing sparse spanners in weighted graphs. *Random Structures & Algorithms*, 30(4):532–563, 2007.
3. 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*, pages 273–284. ACM, 2013.
4. Ruben Becker, Andreas Karrenbauer, Sebastian Krinninger, and Christoph Lenzen. Near-Optimal Approximate Shortest Paths and Transshipment in Distributed and Streaming Models. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 91. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.
5. Soheil Behnezhad, Mahsa Derakhshan, and MohammadTaghi Hajiaghayi. Brief Announcement: Semi-MapReduce Meets Congested Clique. *arXiv preprint arXiv:1802.10297*, 2018.

6 Keren Censor-Hillel, Michal Dory, Janne H Korhonen, and Dean Leitersdorf. Fast Approximate Shortest Paths in the Congested Clique. In *Proceedings of Symposium on Principles of Distributed Computing*. ACM, 2019.

7 Shiri Chechik. Approximate distance oracles with constant query time. In *Proceedings of the Forty-Sixth Annual ACM Symposium on Theory of Computing (STOC)*, pages 654–663. ACM, 2014.

8 Edith Cohen. Polylog-time and near-linear work approximation scheme for undirected shortest paths. *Journal of the ACM (JACM)*, 47(1):132–166, 2000.

9 Michael Elkin and Ofer Neiman. Hopsets with constant hopbound, and applications to approximate shortest paths. In *Foundations of Computer Science (FOCS), 2016 IEEE 57th Annual Symposium on*, pages 128–137. IEEE, 2016.

10 Michael Elkin and Ofer Neiman. On efficient distributed construction of near optimal routing schemes. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, pages 235–244. ACM, 2016.

11 Stephan Friedrichs and Christoph Lenzen. Parallel metric tree embedding based on an algebraic view on moore-bellman-ford. *Journal of the ACM (JACM)*, 65(6):43, 2018.

12 Michael T Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, searching, and simulation in the mapreduce framework. In *International Symposium on Algorithms and Computation*, pages 374–383. Springer, 2011.

13 MohammadTaghi Hajiaghayi, Silvio Lattanzi, Saeed Seddighin, and Cliff Stein. MapReduce Meets Fine-Grained Complexity: MapReduce Algorithms for APSP, Matrix Multiplication, 3-SUM, and Beyond. *arXiv preprint arXiv:1905.01748*, 2019.

14 Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. A deterministic almost-tight distributed algorithm for approximating single-source shortest paths. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pages 489–498. ACM, 2016.

15 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*, pages 938–948. SIAM, 2010.

16 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*, pages 391–410. Society for Industrial and Applied Mathematics, 2015.

17 Christoph Lenzen and Boaz Patt-Shamir. Fast routing table construction using small messages. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 381–390. ACM, 2013.

18 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.

19 Manor Mendel and Assaf Naor. Ramsey partitions and proximity data structures. In *47th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 109–118. IEEE, 2006.

20 Mihai Patrascu and Liam Roditty. Distance Oracles beyond the Thorup–Zwick Bound. *SIAM Journal on Computing*, 43(1):300–311, 2014.

21 Mihai Patrascu, Liam Roditty, and Mikkel Thorup. A new infinity of distance oracles for sparse graphs. In *Proceedings of the 53rd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 738–747. IEEE, 2012.

22 D. Peleg. *Distributed Computing: A Locality-Sensitive Approach*. Society for Industrial and Applied Mathematics, 2000. doi:10.1137/1.9780898719772.

23 Tim Roughgarden, Sergei Vassilvitskii, and Joshua R Wang. Shuffles and circuits (on lower bounds for modern parallel computation). *Journal of the ACM (JACM)*, 65(6):41, 2018.

24 Atish Das Sarma, Michael Dinitz, and Gopal Pandurangan. Efficient distributed computation of distance sketches in networks. *Distributed Computing*, 28(5):309–320, 2015.

25 Mikkel Thorup and Uri Zwick. Approximate distance oracles. *Journal of the ACM (JACM)*, 52(1):1–24, 2005.

26 Christian Wulff-Nilsen. Approximate distance oracles with improved query time. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 539–549. SIAM, 2013.